

Model learning: a survey of foundations, tools and applications

Shahbaz ALI^{1,2}, Hailong SUN^{1,2,3}, Yongwang ZHAO (✉)^{1,2,4}

¹ Beijing Advanced Innovation Center for Big Data and Brain Computing Beihang University, Beijing 100191, China

² SKLSDE, School of Computer Science and Engineering, Beihang University, Beijing 100191, China

³ School of Software, Beihang University, Beijing 100191, China

⁴ School of Cyber Science and Technology, College of Computer Science, Zhejiang University, Hangzhou 310058, China

© Higher Education Press 2021

Abstract Software systems are present all around us and playing their vital roles in our daily life. The correct functioning of these systems is of prime concern. In addition to classical testing techniques, formal techniques like model checking are used to reinforce the quality and reliability of software systems. However, obtaining of behavior model, which is essential for model-based techniques, of unknown software systems is a challenging task. To mitigate this problem, an emerging black-box analysis technique, called *Model Learning*, can be applied. It complements existing model-based testing and verification approaches by providing behavior models of black-box systems fully automatically. This paper surveys the model learning technique, which recently has attracted much attention from researchers, especially from the domains of testing and verification. First, we review the background and foundations of model learning, which form the basis of subsequent sections. Second, we present some well-known model learning tools and provide their merits and shortcomings in the form of a comparison table. Third, we describe the successful applications of model learning in multidisciplinary fields, current challenges along with possible future works, and concluding remarks.

Keywords model learning, active automata learning, automata learning libraries/tools, inferring behavior models, testing and formal verification

1 Introduction

Developing a bug-free software is a challenging job, and ensuring its correctness is more challenging. The likelihood of occurring errors in systems rises with the increase of scalability and functionality. The heavy use of unspecified third-party components, for rapid development and reducing time-to-market, can also result in under-specified and erroneous systems. Unfortunately, bugs are common and propagate among these components. Integration testing of the overall system, developed by unspecified black-box components, becomes a challenging job. There are numerous cases where software bugs result in a disastrous loss of money, time, or even human life. Some of incidents, regarding notorious bugs, including failure of Patriot

missile [1] during the Gulf war (due to software error), the crash of the Ariane 5.01 maiden flight [2] (due to an overflow), the loss of Mars orbiter [3] (due to a unit error), Therac-25 radiation therapy machine [4] (software error), and Pentium FDIV [5] (design error in floating-point division unit) are a few well-known examples highlighting the fact that life-depending, mission-critical and safety-critical systems can be unsafe.

The profusion and diversity of these bugs have generated the requirements for promising techniques to avoid, detect, and correct the flaws. The versatile model-based techniques, including model-based testing (MBT) and model checking, are effective methods to enhance the quality and reliability of software systems by removing the bugs. The existence of behavioral models of systems under test (SUTs) is the pre-requisite for the application of model-based techniques. These techniques are best-fit for the scenarios where behavioral models are readily available either from requirement specifications or by applying the program analysis techniques to source code directly [6–8]. However, these techniques are not suitable for software systems, which are generally distributed without access to source code and have limited documentation (or the available documentation is outdated due to system evolution). As the construction of formal models is the fundamental problem for black-box software components, so we must consider some alternative approaches from the domain of software reverse-engineering.

In recent years, a new technique called *Model Learning* (a.k.a automata learning) has emerged as an effective technique for learning the models of black-box hardware and software systems. It complements testing and verification approaches by providing accurate and reliable models of systems under learning (SULs). Generally, model learning approaches can be categorized into active and passive learning. The passive learning algorithms learn the behavioral models of SUL from the available set of positive and negative traces (training data) [9, 10]. In active learning, there is a continuous interaction between the learning algorithm (learner) and SUL. It is a *dynamic analysis* technique and infers the behavioral models of black-box SUL by performing experiments (tests) on it. By applying this process of experimentation repeatedly, a model is approximated, which represents the complete and run-time behavior of SUL. It is the test-based and counterexample-driven approach for learn-

ing models of real-world black-box software systems fully automatically. In this survey paper, we restrict ourselves to “*Active Learning approach*” that provides behavioral models of “*Black-Box systems*” in the form of “*State Diagrams (or Automata)*”.

The learned models are highly effective and can benefit model checking approaches [11–13], model-based testing [14, 15], composition of components [16], checking correctness of API usage [17], integration testing of black-box components [18–22], and to facilitate reuse [23]. Model learning techniques can be used to re-engineer (or reverse-engineering) some modules of legacy systems, and to test the modified version of a software component (regression testing).

The core part of model learning is the learning algorithms that provide the models of realistic systems. Besides learning algorithms, testing (for validation checking), and optimal counter-example handling algorithms have also been developed. For practical applications, we need robust and versatile implementations of learning algorithms along with surrounding infrastructure, which is vital for quick assembly of learning setups. To satisfy these needs, some progressive work in the direction of learning libraries and tools has been accomplished. The primary objective of these learning tools is to provide a framework for research on automata learning and their applications in real life.

The rest of the paper is structured as follows: the next section lays out the background knowledge of model learning. Section 3, covers the key concepts about the foundations of model learning along with a table that highlights the comparison of different learning algorithms. Section 4 presents the progress made in the development of different model learning libraries and tools. Section 5, describes the successful applications of model learning in various domains. Section 6 summarizes the current achievements, challenges, and possible future trends in the field of model learning.

2 Background

In this section, we describe some background knowledge of model learning. It includes behavior models, behavioral model inference approaches and their evaluation, learning frameworks, and scope of our survey.

2.1 Behavior models

In recent years, model-driven engineering (MDE) has attracted much attention as an effective software development methodology. In the MDE approach, requirement specifications, system behavior, functionality, development, and testing activities of systems can all be represented in terms of models. These models are very useful and can be used as a way of communication between different stakeholders like customers, designers, developers, testers, product managers, and end-users of the software. Moreover, these models can also be used for the generation of implementations (semi) automatically. Examples of models may include a function, a relation, a logic formula, a directed graph, stochastic models, state diagrams (or automata), algebraic models, logic programs, etc.

A state model (or transition-based model), which is also the focus of our survey paper (behavioral models have been discussed in Section: 3), can be used to express the essential behavior of many realistic systems like security protocols, net-

work protocols, and control software of embedded systems. Behavior models are the basis for development and verification techniques, for example, model-based testing (model-based test case generation), model checking, service composition, and controller synthesis.

2.2 Model learning (automata learning)

The automatic construction of state-transition models through observations is called automata learning (aka regular inference). We frequently construct and use such models in our daily life. For example, to learn the behavior of a computer program or device (say remote control of a TV), we press all the available buttons and observe its responses. In this way, we construct a mental model of that program or device. Using this saved mental model, we operate that program/device easily without consulting its manual. Now, the main challenge is to let computers perform a similar kind of learning tasks in a systematic way for real-life systems having more states.

The problem of automata learning has been studied in the literature for decades and remains a hot topic of research. In recent years, it has gained much attention in software engineering tasks, especially in testing and verification. First of all, Moore [24] highlighted the problem of learning finite automata in 1956. Since then the problem of learning automata has been studied by different research groups under different names like grammatical inference [18], regular inference [25], regular extrapolation [26] or active automata learning [27] and protocol state fuzzing [28]. In recent years, researchers have used the term of *Model Learning* in resemblance with model checking technology and highlight the emergence of model learning as an effective bug-finding technique by inferring models of black-box hardware and software components [29].

Ideally, behavior models should be produced during early design phases, but in practice, they are omitted due to various reasons. To mitigate this problem of non-existing or outdated behavioral models, model-learning techniques have been developed which infer models of systems automatically. Model learning is a complementary technique for model checking because finite-state models are analyzed in model checking, whereas in model learning models of software and hardware systems are constructed by providing inputs and observing outputs. Moreover, research works by Peled et al. [30], and Steffen et al. [25, 26] have brought model learning and formal method techniques close to each other, particularly model checking and model-based testing.

2.3 Model learning approaches

Depending on the utilization of source code, model learning is either a white-box or a black-box. Further, model learning approaches can be divided mainly into active and passive learning, each having its characteristics. In the following paragraphs, we briefly discuss the two learning approaches.

2.3.1 Passive learning

In passive learning, there is no interaction between the learner and SUL. The passive learning algorithms, also called offline algorithms, learn the behavioral models of SUL from the available set of positive and negative traces (training data) [9, 10, 31–33]. Positive traces belong to the target language of

SUL and negative traces do not. The behavior of a system, for a specific period, is recorded in the form of traces in a log file. Every trace consists of input symbols representing the stimuli the SUL is exposed to, and its output symbols representing the system's reaction to the input symbols. Passive automaton learning is a form of supervised learning, since we are given pairs of input symbol (queries) and output symbols (observations), and asked to fit an optimal automaton structure to them [34].

The main problem with passive learning is that the quality of the learned model depends upon the recorded traces. For a log file having missed or unobserved behavior, it is difficult to infer a conclusion, and hence the resulted model becomes erroneous due to this unobserved behavior. There exists a solution to this problem, which is to request additional behavioral traces, as per requirement, from the target system. This strategy has been incorporated into *active learning* approach.

2.3.2 Active learning

In active learning, there is a continuous interaction between the learner and SUL. The active learning algorithms, also called on-line algorithms, learn the behavioral models of SUL by performing experiments (tests) on it. By applying this process of experimentation repeatedly, a model that represents the complete behavior of a software component is approximated. Like passive learning, active automaton learning is also a supervised learning problem since learning is being carried out under the supervision of a qualified teacher (SUL) [34].

2.3.3 Evaluation of learning techniques

Recently, various learning algorithms have been developed for learning behavioral models of software components. It is difficult to make a decision about which algorithm is better than others. Every algorithm has its own working environment, which may perform better than other algorithms in specific settings. A few available benchmarks are not so diverse that they can make a reliable comparison. One solution is to develop a framework that can evaluate the learning algorithms and provides an opportunity to have a look at the weaknesses and strengths of other participant's algorithms. Finite state machines or grammar learning competitions like *Tenjinno* that is a machine translation competition [35], *Omphalos* is a context-free grammar learning competition [36], *STAMINA* [37] and *ZULU* [38] competitions are some frameworks that can help in evaluating learning algorithms. If such competition ends with no winner, these are still useful in the sense that all the participants can have a look at the weaknesses and strengths of other's learning techniques.

2.4 Learning frameworks

The three well-known learning notions are Angluin's learning model [39], Gold's learning model [40], and PAC learning model [41]. In this subsection, we give a brief description of the Angluin's model that forms the basis of active model learning (which has been described in subsequent sections). We shall also give a short overview of the other two learning notions, i.e., Gold's learning model and PAC learning model.

Angluin's learning model is a well known formal learning model developed by Dana Angluin in 1987. It is one of the most

important models of computational learning theory. A concept of *minimally adequate teacher (MAT)* was introduced by Angluin, and the learning process can be viewed as a game. In the so-called MAT framework, the construction of models involves a "*learner*" and a "*teacher*", where the role of the learner (a learning algorithm) is to learn an unknown model by putting the queries to a teacher/oracle. The teacher knows all about the SUL, but the learner knows only the input/output alphabets of the SUL. For learning an unknown automaton \mathcal{M} , the learner poses queries and then tries to construct an unknown automaton (by using received responses of queries) whose behavior matches with the target automaton. This kind of learning is called *query learning* (or *learning via queries*) and the objective is referred to as *exact identification*.

In Gold's learning model, Gold [40] introduced a learning notion in 1967 called *identification in the limit or EX identification*. For this learning notion, there exists a framework [42] which has been extended [43] to study *query learning*. However, in this kind of learning, the main issue is not the learnability within a given amount of time but the learnability within a finite amount of time. We can say that the framework is not suitable for resource bounded-learning but suitable for recursion theoretical learning [44].

In probability approximately correct (PAC) learning model, for resource bounded-learning, Valiant [41] introduced the PAC learning model, which is polynomial-time bounded-learning. Due to much research work in this field [45, 46] there exists now a PAC learning framework [47]. However, the learning approach in this framework is "passive learning", which will be discussed in the next section. Angluin [48] also studied and discussed this notion of learning style in their research work.

2.5 Objective and scope of survey paper

After having some background knowledge of behavior models and their significance, black-box systems, active model learning, and passive model learning, now in the subsequent sections, we restrict ourselves and focus only on *active learning* of *black-box* software systems in the form of *state transition models (automata)*. We shall cover foundations of active model learning, then discuss the developments in model learning tools, and finally describe the successful applications of model learning in multidisciplinary fields followed by open challenges and possible future directions.

3 Foundations of model learning

This section covers the key concepts that form the foundation of model learning. It includes basic framework, a basic algorithm for active learning, various modeling formalism, learning and testing algorithms, the role of abstraction and mapper, data structures, learning complexity, and a running example to understand the working of model learning. At the end of the section, it summarizes the section in tabular form.

3.1 Basic framework for active learning

In model learning, behavioral models can be inferred using different learning approaches, including performing tests (queries), mining system logs (from traces), or analyzing source code. Among these approaches, active automata learning (model learning) has gained great attention, and most

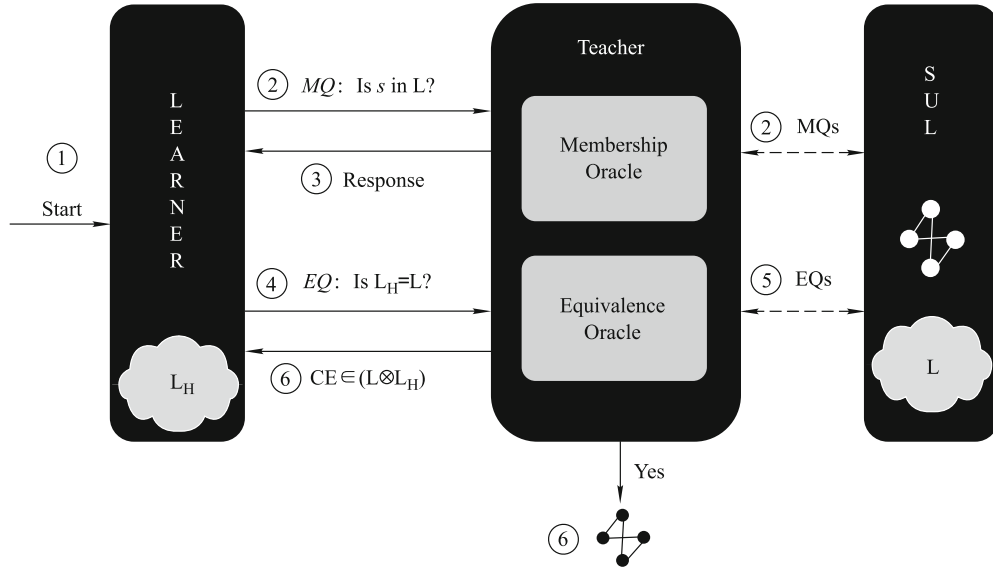


Fig. 1 General learning process with basic MAT framework

of the learning algorithms in this approach follow Angluin's MAT framework (i.e., learning actively by posing queries). An overview of a general flow of the active learning process using MAT framework is shown in Fig. 1.

The active learning framework usually makes the following assumptions for learning models. First, it assumes that the SUL can be modeled as a finite automaton. Second, the learner knows the fixed I/O alphabets of the target model. Third, it also assumes that there is an oracle (a.k.a., teacher) who has complete knowledge about SUL. Additionally, the learner can *reset* the teacher at any point. A reset command is used to bring the SUL back to its initial state. Reset is important because active learning requires MQs to be independent [49]. However, now there exist two approaches that infer the models of systems without resetting. The first one is proposed by Rivest et al. [50], which is based on homing sequence, while the second approach has been proposed by Groz et al. [51, 52], which is based on characterization sequences.

• Basic algorithm for active learning

In the context of active learning, Dana Angluin proposes a query learning algorithm [39], L^* for inferring DFA, which describes regular sets. The learning of DFA with the L^* algorithm is a typical example of polynomial-time learning via queries. Many efficient active learning algorithms have been designed and developed since then, and most of them follow the Angluin's approach of MAT framework.

The main components of the L^* algorithm have been shown in Fig. 2. Learner poses two types of queries:

- *Membership Queries (MQs)*: With MQs the learner asks whether the input sequence belongs to the target system. The answer is “Yes” if it belongs to, otherwise “No”.
- *Equivalence Queries (EQs)*: With EQs the learner asks whether the hypothesized/conjectured automaton \mathcal{H} is correct i.e., $\mathcal{H} \approx \mathcal{M}$. The teacher who knows about the system completely answers “Yes” if this is the case otherwise it answers “No” and provides a counterexample that differentiates \mathcal{H} from \mathcal{M} such that $\mathcal{H} \neq \mathcal{M}$.

The learner records the received responses (0 or 1) into a data structure called an *observation table* (section: 1). In this learning process when the table becomes *closed* and *consistent* then learner builds an automaton called conjecture/hypothesis with the help of a filled observation table.

Now, to check whether this hypothesis is equivalent to the model of black-box SUL, the learner depends upon the existence of some oracle. To check the validity, the learner uses EQs (*yes/counterexample query*). If the black-box SUL is not equivalent to the hypothesis, then the oracle generates a counterexample, which is a string from the input set that is accepted by the black box and rejected by hypothesis and vice-versa. By analyzing the counterexample in an intelligent way, the learner refines the observation table and constructs an improved

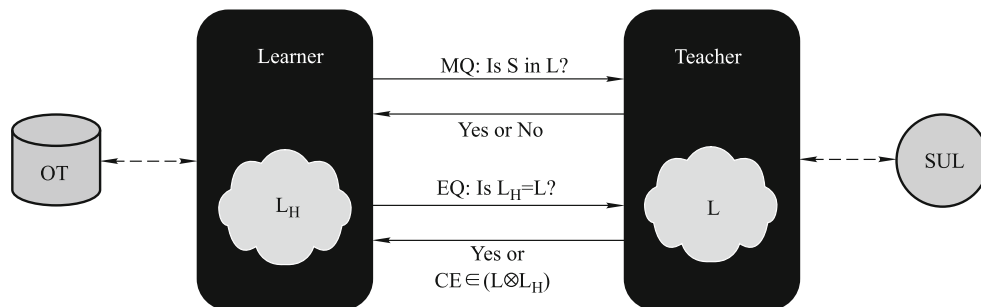


Fig. 2 Components of L^* algorithm

version of the hypothesis. The whole process continues in a loop until the learner returns an automaton whose behavior matches with the SUL.

For a regular set, L^* returns a minimal DFA \mathcal{M} provided that the reply of every query is given correctly w.r.t the target regular set. Moreover, L^* obtains \mathcal{M} in polynomial time so, the regular set class, which is represented by DFA, is polynomial-time learnable.

3.2 Learned models

Having a formal representation (model) of a SUL is the prerequisite for model-based verification techniques. In the following subsections, we give a brief overview of the well-known automata models which are being used in the model learning paradigm.

3.2.1 DFA and mealy models

A simple and well-known modeling object in language theory is DFA (deterministic finite automaton) that models and recognizes regular languages [53]. In the seminal work of Angluin, a DFA representing an unknown regular language was learned [39]. However, DFAs are not suitable candidates for modeling realistic systems (reactive systems) because of their limited expressiveness. To model reactive systems (input/output based systems), *Mealy model* is a better option that takes inputs, processes them, and returns outputs. These systems interact with the environment and work in a cyclic loop.

Various algorithms based on Angluin's approach model the systems in the form of Mealy machines [19, 49, 54] (see Table 1). A slightly different approach to model systems in the form of Mealy machines is suggested by [22, 55] that replaces the MQS by output queries and record the output responses directly into a data structure known as *Observation table (OT)*. Shahbaz et al. [56] used a more expressive version of the Mealy machine called parameterized finite state machine (PFSM), which is more expressive than simple Mealy models. Aarts et al. [57] learned Mealy models of bounded retransmission protocol (BRP) implementation and used equivalence checking to compare with reference specifications. Learning algorithms and their applications regarding DFA/Mealy machines have been discussed in Table 1 and Table 3, respectively.

3.2.2 LTS and IOTS models

Hagerer et al. [26] proposed a technique called *regular extrapolation* to infer the models of *Computer telephony integrated (CTI)* in the form of LTS with inputs and outputs. Walkinshaw et al. [58] demonstrated a reverse-engineering technique on a real Erlang FTP client implementation. They inferred the models of implementations in the form of LTS state machines. Hungar et al. [18] adapted the L^* algorithm to learn the models in the form of LTS with inputs and outputs. In their setup, they made some optimizations based on domain-specific knowledge to reduce the number of membership queries. Groz et al. [59] developed an approach to verify a modular system by inferring the models of its components. They inferred the models in the form of Input-Output Transition System (IOTS) to detect the errors and compositional issues in modular systems.

3.2.3 EFSM model

To model realistic systems, like software protocols, extended

finite state machine (EFSM) and their variants are more suitable options. EFSM can model data flow as well as the control flow of a software module since input and output symbols can carry data values. A specific restricted model on EFSM formalism is the register automata, in which finite control structure is combined with variables, guards, and assignments. Various dialects of EFSMs are used successfully in ConformiQ Atronic model-based testing tool [60] and in the software model checking [61]. Some work to generalize learning algorithms to richer models (like EFSMs), where data values can be communicated, processed, and stored, have been accomplished in [62–66].

3.2.4 Symbolic finite automata (SFAs)

The classical automata theory has two basic assumptions: (1) a finite space, and (2) a finite alphabet. However, there are many realistic applications that use values from the infinite domain for individual symbols. Sometimes the choice of classic automata can be problematic in spite of the finite alphabet: for example, a DFA modeling a language over the UTF-16 alphabet needs 2^{16} transitions out of each automation state. Symbolic Finite Automata (SFA) are finite-state automata in which transitions carry first-order predicates instead of concrete symbols. In SFAs, the alphabet is given by Boolean algebra, having an infinite domain. Large character sets, like UTF, can be represented easily by using the SFA model. For example, a transition of the form $0 - [a - z] \rightarrow 1$ represents a transition when we move from state 0 to state 1 with all symbols from the interval $[a - z]$. In general, SFAs are more expressive and outperform their classical counterparts for large alphabets.

Algorithmic work for learning models in the form of SFAs, using Angluin style learning (MAT model), has been carried out in [67–70]. From the application point of view, symbolic finite-state transducers (SFTs) can be used to describe and analyze *sanitizers* (e.g., HtmlEncoder, UrlEncoder, CssEncoder), to check domain equivalence of SFTs [71]. Another application of SFAs is in the domain of *Regex* (regular expression) processing [72]. Besides, Argyros et al. [73] presented SFADiff, a black-box differential testing framework, which is based on (SFA) learning to find the differences between programs having similar functionality. The work can be used for fingerprinting or creating evasion attacks against security software like Web Application Firewalls (WAFs).

3.2.5 Other computational models

Some other modeling formalisms for which learning algorithms have been developed include: Timed automata (to model clock time bounds on transitions and useful for real-time embedded systems) [74, 75], probabilistic and non-deterministic automata (useful for autonomous and large-scale distributed systems for which we may have limited knowledge; useful for probabilistic model-checking) [76–79], hybrid automata (combining discrete and continuous state transitions into one model; useful for cyber-physical systems) [80], Petri-nets (to represent systems with explicit parallel state) [81], Buchi automata [82] and pushdown automata [83].

3.3 Learning and testing algorithms

In this subsection, we briefly discuss learning and testing algorithms, which play vital roles in active model learning.

3.3.1 Learning algorithm (Learner)

Learning algorithms are the backbone of automata learning techniques for obtaining models of realistic systems. These algorithms are also playing a key role in various domains of computer science, including AI, data mining, neural networks, robotics, geometry, pattern recognition, natural language processing, and particularly in verification and testing. These algorithms can be differentiated based on data structures, modeling formalism, and the learning approaches (active or passive) they use.

Some well-known learning algorithms are L^* [39], NL^* [84], ADT, DHC [85], Maler & Pnueli [86], Shahbaz & Groz [55] and Suffix1by1 [87]. Kearns and Vazirani [88] improved the L^* algorithm by replacing the table structure with a discrimination tree. Rivest and Schapire [50] also improved the L^* algorithm by introducing an efficient counterexample handling mechanism. Currently, the TTT algorithm designed and developed by Isberner et al. [27, 89] is believed to be the best algorithm among the available model learning algorithms. This algorithm was designed and developed by overcoming the inefficiencies in the algorithms proposed by Kearns & Vazirani and Rivest & Schapire. TTT algorithm efficiently eliminates the excessively long discrimination trees, which may be resulted while processing long counterexamples.

3.3.2 Testing algorithm (Equivalence Oracle)

Peled et al. [30] and Groce et al. [90] highlighted the fact that MAT framework can be utilized to infer the models of black-box hardware and software components. In an active learning process, the learner's job is to construct hypotheses by posing queries to SUL, whereas the job of an equivalence oracle is to validate the constructed hypotheses. In general, an equivalence oracle does not exist, and designing it for black-box SULs is a challenging task because there is no direct method to make a comparison between the structures of black-box SULs and learned hypotheses. Black-box equivalence checking algorithms, from the domain of MBT, are SUL's structure independent and can approximate *equivalence oracles* by generating test suites as shown in Fig. 3. In validating a software implementation, for example, "*random sampling oracle*" can be used to explore the learned hypotheses and implementation (SUL) to search for discrepancies.

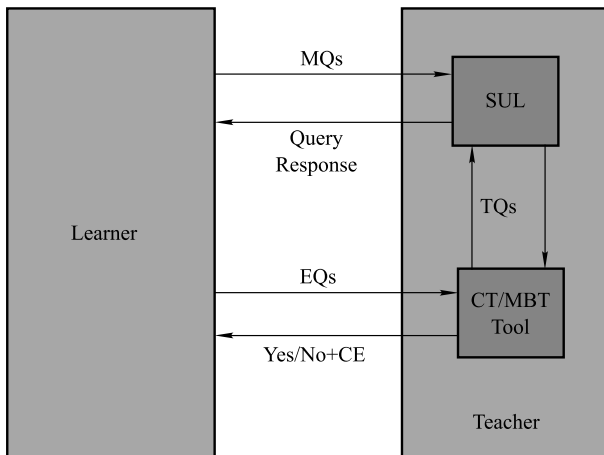


Fig. 3 Active automata learning in a black-box reactive system [29, 91]

As a testing tool can only send a finite number of testing queries (TQs) and it may fail to find any counterexample so we cannot say with certainty that the learned model is the correct representation of the target system. However, by assuming a bound on the number of states of the SUL model, we can have a finite and complete conformance test suite [92]. In recent years, some efforts to predict the quality of learned models have been made by [93, 94].

A wide variety of algorithms, which act as equivalence oracles, have been proposed for different classes of models. Automata learning libraries, for example, LearnLib (section: 2), provide the implementations of various testing algorithms (equivalence oracles). These algorithms include state cover set, transition cover set, Random words, W-Method [95], Random walk, Wp-Method [96], W-method randomized, Wp-method randomized, UIO method [97] and UIOv method [98].

3.4 Model learning using abstraction

During the model learning process, the formation of a hypothesis is a relatively easy task, but its validation using conformance testing becomes a challenging task for large input alphabet. Let, " n " be the number of states present in learned hypothesis (H), " \hat{n} " be the states present in SUL, and if " p " be number of inputs then in the worst scenario, we require to run test sequences of $\hat{n} - n$ inputs that is, $p^{(\hat{n}-n)}$ possibilities [29, 92]. So, there is a requirement for some strategies that can reduce the number of inputs. One solution is to use *abstraction* technique.

Model-based verification and validation tools, for example, ConformiQ qtronic [60], deal with the data influence on control flow. But, model inference techniques, which employ learning algorithms with finite alphabet size, suppress the message parameters, although these parameters have a significant influence on control flow in many communication protocols. The message parameters can be flags, configuration parameters, sequence numbers, the identities of an agent or a session, etc.

To apply model learning techniques to realistic systems, which have large inputs and outputs, *abstraction* plays a key role. Aarts et al. [99, 100] proposed a framework that modifies the learning technique so that it can have data parameters in messages and states for the generation of components with large message alphabets. Mapper is a software component that plays the role of abstraction and is placed between learner and teacher. The learner sends abstract messages to the mapper component that transforms them into concrete messages and forwards them to the SUL. The mapper component converts the concrete responses of the SUL back to abstract messages and sends it to the learner, as illustrated in Fig. 4. A formal model of the abstracted interface is learned by model learning. To obtain a faithful model of a SUL, the abstraction process can be reversed.

Cho et al. [101] applied a model learning technique on botnet command and control protocols for inferring models. They place an emulator/mapper component between learning software and botnet servers. Mapper component concertizes the abstract alphabet symbols into valid network messages and forwards them to botnet servers. On receiving responses from the servers, the mapper/emulator does the reverse of it, i.e., it converts response messages into abstract output alphabet and for-

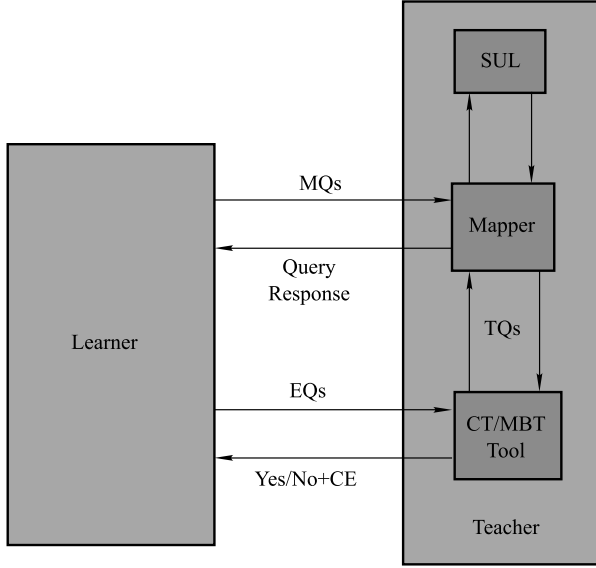


Fig. 4 Model learning with mapper component [29]

wards to the learner. To reduce the number of inputs, Chalupar et al. [102] used a different approach for inferring models. They merge many input actions that have a specific order into a single high-level action.

Researchers normally define abstraction manually. Vaandrager [103] proposed a technique that creates an abstraction component automatically for the richer classes of models, e.g., EFSMs. In such modeling formalism, the equality of data parameters can be tested, but operations on data are not allowed [104, 105]. Their approach makes use of the counterexample-guided abstraction refinement (CEGAR) notion. In the CEGAR concept, whenever the current abstraction becomes very coarse and causes non-determinism, the employed abstraction is refined automatically. Vaandrager [103] compared this technique with the work done by Howar et al. [62, 64] on register automata model.

3.5 Data structures

Model learning algorithms normally vary in two aspects: (1) the data structures used for storing responses of queries and realizing the black-box abstraction, (2) and how learning algorithms handle counterexamples. The basic learning algorithm L^* and its variants used two types of data structures: *observation table* and *discrimination tree*.

3.5.1 Observation table

An observation table \mathcal{T} is the most well-known data structure, and it was introduced first by Gold [106]. Later on, Angluin used this data structure in her seminal algorithm L^* for organizing the information collected during the interaction with the SUL. Different learning algorithms have used variants of observation table for other modeling formalisms.

Let Σ be the input alphabet, and Ω be the corresponding output alphabet of the SUL model. The problem of learning model (i.e., constructing Q, q_0, Σ, Ω) from *queries* and *observations* is called *black-box learning*. The observation table \mathcal{T} may be a dynamic two-dimensional array that is characterized by upper and lower parts. The entries in the table are the values from the

output alphabet set Ω . The rows and columns of this table are indexed by strings over Σ^* . Learning the model of a SUL is an incremental process, so the table is allowed to grow accordingly over time. Formally, it can be represented as a triple over an alphabet Σ , by (S, E, T) . Here, S and E are sets of strings over Σ which are non-empty and finite. During the learning process, the table is updated continuously by the learner, and it can be visualized by three distinguished indexing sets: (1) $S \subseteq \Sigma^*$ is *prefix-closed* set of input strings. The values in this part (upper left of \mathcal{T}) are used to represent the states of minimal state-model, (2) $S.\Sigma$ is also a *prefix-closed* set of input strings. The values in this part (lower-left of \mathcal{T}) are used in building transitions of state-model, and (3) $E \subseteq \Sigma^*$ is a *suffix-closed* set of input strings. The rows and columns of \mathcal{T} are indexed by the sets $(S \cup S.\Sigma)$ and E respectively.

In the above expression (S, E, T) of an observation table, T is a finite function which is defined (for Angluin's Lerner L^*) as $\mathcal{T} : ((S \cup S.\Sigma) \times E) \rightarrow \{0, 1\}$. For any row $s \in (S \cup S.\Sigma)$, column $e \in E$, the function $T(s, e)$ represents the corresponding cell in table. The value of $T(s, e)$ is "1" if the string $s.e$ is accepted by the target SUL and "0" in other case. In order to construct an automaton from the filled observation table, \mathcal{T} must fulfil the properties of *closed* and *consistent*. Let, $\mathcal{T} : ((S \cup S.\Sigma) \times E) \rightarrow \Omega$ is a two dimensional observation table. (1) \mathcal{T} is **closed** if $\bar{s} \in S$ and $\sigma \in \Sigma$, there exists $\bar{q} \in S$ such that the two rows $\mathcal{T}[\bar{s}.\sigma]$ and $\mathcal{T}[\bar{q}]$ becomes equal i.e., $\mathcal{T}[\bar{s}.\sigma] = \mathcal{T}[\bar{q}]$. (2) \mathcal{T} is **consistent** if whenever $\bar{p}, \bar{q} \in S$ such that $\mathcal{T}[\bar{p}] = \mathcal{T}[\bar{q}]$ then for $\forall \sigma \in \Sigma$ we have $\mathcal{T}[\bar{p}.\sigma] = \mathcal{T}[\bar{q}.\sigma]$.

3.5.2 Discrimination tree

The second data structure used by many learning algorithms is a discrimination tree (DT) which is a decision tree for determining the equivalence of states. It is a classification scheme for distinguishing between states and first introduced by Kearns & Vazirani [88].

Figure 5(b) shows a discrimination tree that is obtained by applying the *Observation Pack* algorithm on SUL having formal behavior shown in Fig. 5(a). Here, leaves represent the states of hypothesis, and discriminators labeled the inner nodes. Further, every node holds two children, i.e., 0-child (represented by the dashed line) and 1-child (represented by a solid line). Two well-known operations which are used to get information from a discrimination tree are: (1) "*Sifting*" and (2) computing "*lowest common ancestor (LCA)*". The "*sifting*" explores the tree for classification, and "*LCA*" emphasizes the separation of classes presented in a tree. For further details about the discrimination tree data structure, we refer the interested readers to [27, 107].

3.6 A running example

In the previous sections, we have discussed some key concepts of model learning, like learning algorithms (learners), testing algorithms (acting as equivalence oracles), various kinds of learned models, and data structures. To understand the operations of model learning, we have prepared a working example. However, due to space limitations, we have placed this running example on our repository "(bazali/Working-Examples-of-Model-Learning)" created on Github. In this example, we

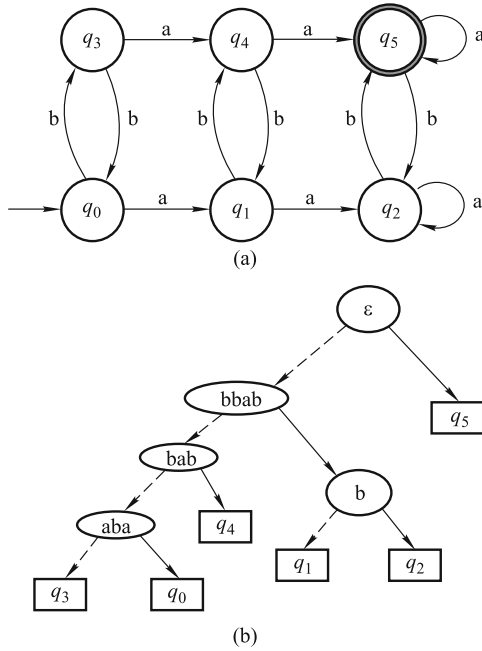


Fig. 5 A simple DFA with discrimination tree. (a) A simple DFA; (b) discrimination tree using *observation pack* algo [107]

have learned a DFA model that accepts a regular language with an even number of 0's and an even number of 1's (the numbers of 0's and 1's are not required to be equal). Moreover, in the example, we have used a well-known Angluin's L* algorithm for posing MQs and EQs, and to handle counterexample, we used the approach of Rivest & Schapire to refine the hypothesis.

For more running examples, we refer the interested readers to [29, 49, 108–115]. These running examples employ L* and its variants as learners and demonstrate the operations (such as, posing of MQs and EQs, handling of counter-examples, saving responses to data structures, etc.) of active model learning process.

3.7 Learning complexity

A learning algorithm's complexity is normally calculated in terms of the required number of membership and equivalence queries. It is due to the fact that the execution of membership queries requires interaction with the SUL, and it will take some time. The observation table or other data structures like reduced observation table or discrimination tree for queries and responses need to allocate memory resources. Let the size of the alphabet Σ , is denoted by $|\Sigma|$, " n " be the total number of states in the target minimal state model of SUL, size of input alphabet I is $|I|$, and " m " be the longest counterexample returned by the oracle. For Angluin's algorithm, and the algorithms using structures like discrimination tree or reduced observation table, the upper bound for the required number of equivalence queries is " n ".

Now we discuss the upper bound for membership queries, which is not so simple as the case of equivalence queries. The upper bound on algorithms like Angluin's that uses *observation table* data structure is $O(|\Sigma|mn^2)$ [55]. The algorithms like Rivest & Schapire [50] that use *reduced observation table* data structure (a reduced version that stores a smaller portion of queries and their responses) has upper bound on membership

queries as $O(|\Sigma|n^2 + nm)$. The third type of algorithms like Kearns & Vazirani [88] use *binary discrimination tree*, a completely different data structure for storing information. Such a category of algorithms has upper bound on membership queries as $O(|\Sigma|n^2 + n \log m)$. The upper bounds on membership queries depend on responses to these queries whether they are stored or not (not saved in case of discrimination tree but saved on the other two cases) and how many these membership queries are posed before creating a hypothesis [116, 117]. Among these three categories of algorithms, Angluin's algorithm normally poses more queries to build a model, and hence it collects more information. Due to this reason, Angluin's algorithm produces less false hypotheses and thus fewer equivalence queries. For these three categories of algorithms, the upper bound for equivalence queries is, however, the same, i.e., n .

Berg et al. [118] analyzed the performance of the Angluin's algorithm by considering randomly generated automata and real-world examples. They studied the impact of alphabet size $|\Sigma|$ and the total number of states n on the required number of membership queries (MQs). As complexity also depends upon the length of counterexample, so handling counterexamples efficiently [21, 50, 55, 86, 119], several algorithms have been proposed.

3.8 Benchmarks

Various benchmarks have been used for the evaluation of learning and conformance testing algorithms. Randomly generated machines (as benchmarks) [84, 118, 120, 121] were also used to evaluate testing techniques. However, these benchmarks are small, academic, or randomly generated, which do not properly evaluate the efficiency of techniques. Moreover, the performance of these algorithms on such kind of benchmarks is often different from the performance on models of real systems that occur in practice.

The Radboud University has established a repository "Automata wiki" and shares publicly several well tested and useful benchmarks of state machines that model real protocols and embedded systems. Researchers can use these benchmarks for comparing the performance of learning and testing algorithms. Aarts et al. [122] in his research work compared two established approaches of inferring register automata. The authors have set up a repository (LearnLib/raxm) on Github to place benchmarks regarding register automaton models received from a variety of sources and application domains. They range from explanatory examples that have been discussed in the literature, to manually written specifications of data structures, to models that were inferred from actual systems (e.g., the Biometric Passport and the SIP protocol).

Models provided by Edinburgh concurrency workbench (CWB) can also be used as benchmarks to evaluate learning and conformance testing algorithms. CWB [123] is a tool that is used for analysis and verification of concurrent systems. This tool provides several fabricated finite models for realistic systems, including vending machines, mailing systems, ABP and ATM protocols, etc. All synthetic models provided by CWB have a different number of states and input set size.

3.9 Comparison of learning algorithms

After having some background knowledge of different entities,

Table 1 Comparison of different learning algorithms

ALGORITHMS #	DS	LM	Key features	Ref.
L*	OT	DFA	<i>Implementation:</i> LearnLib; Libalf; AIDE and RALT; <i>Complexity:</i> $O(\Sigma mn^2)$	[39]
L_{col}^* ("Maler/Pnueli")	OT	DFA	<i>Implementation:</i> Libalf; <i>Complexity:</i> $O(\Sigma mn^2)$	[124]
Mealy Machine algo L_M (adaption of L*)	OT	Mealy	<i>Implementation:</i> RALT <i>Complexity:</i> $O(\Sigma mn^2)$	[55]
Mealy Machine algo L_M + (CE handling improvements)	OT	Mealy	<i>Implementation:</i> RALT <i>Complexity:</i> $O(\Sigma mn^2)$	[55, 125]
ADT (Adaptive Discrimination Tree)	DT	FSM Mealy	<i>Implementation:</i> LearnLib	–
DHC (Direct Hypothesis Construction)	-	Mealy	<i>Implementation:</i> LearnLib; AIDE; <i>Data Structure:</i> Direct construction of hypothesis from observations, i.e., without observation table.	[126]
Discrimination Tree	DT	DFA, Mealy, VPDA	<i>Implementation:</i> LearnLib <i>Complexity:</i> $O(\Sigma n^2 + n \log m)$	[88]
Kearns–Vazirani (original)	DT	DFA, Mealy	<i>Implementation:</i> LearnLib, Libalf; <i>Complexity:</i> $O(\Sigma n^2 + n^2 m)$	[127]
Kearns–Vazirani (bin search)	DT	DFA, Mealy	<i>Implementation:</i> LearnLib <i>Complexity:</i> $O(\Sigma n^2 + n^2 \log m)$	[127]
PFSM algorithm L_P^*	OT	PFSM	<i>Implementation:</i> RALT <i>Complexity:</i> $O(\Sigma mn^2)$	[128]
L_1 /Suffix1by1	OT	Mealy	<i>Implementation:</i> AIDE <i>Complexity:</i> $O(\Sigma mn^2)$	[111]
Observation Pack	DT	Mealy	<i>Implementation:</i> AIDE; It Builds upon Rivest&Schapire's algo; <i>Complexity:</i> $O(\Sigma n^2 + n \log m)$	[127, 129]
NL*	OT	NFA	<i>Implementation:</i> LearnLib; Libalf. Learning algorithm learns NFA using MQs and EQs. More specifically, residual finite-state automata (RFSA) are learned	[84]
TTT	DT	DFA, Mealy, VPDA	<i>Implementation:</i> LearnLib. It mitigate the effects of long-counterexamples; <i>Complexity:</i> $O(\Sigma n^2 + n \log m)$	[107]
SL*	SDT	RA	<i>Implementation:</i> RALib	[130]
A* (Symbolic Learning Algo)	OT	SFA, SVPA	<i>Implementation:</i> Symbolic Automata Lib	[71, 131]

like modeling formalisms, data structures, and learning complexity, which are associated with learning algorithms, now we compare them in Table 1. In this table, "DS" represents data structure, "LM" stands for learning model, "OT" is for observation table, "DT" stands for discrimination tree, and "SDT" represents symbolic decision tree.

4 Model learning tools

To make model learning practical, we need efficient implementations of learning algorithms. Besides algorithms, the surrounding infrastructure is also compulsory for the rapid assembling of learning setups. To satisfy these needs for a flexible and comprehensive framework for model learning, different learning libraries have been developed. In the following, we will describe some well known learning libraries/tools that are being used widely in the domain of model learning.

4.1 AutomataLib

AutomataLib is a free, open source library implemented in Java. It was developed at the Dortmund University of Technology, Germany. It supports a variety of graph-based modeling structures. Its main objective is to serve LearnLib, an-

other Java library, which will be discussed in the next section. However, the implementation of AutomataLib is completely independent of LearnLib and can be used for other projects as well. AutomataLib supports some selected classes of NFA, but it mainly focuses on DFA. The current version of AutomataLib (0.8.0) supports the modeling of generic transition systems, DFA, Mealy machines, and advanced structures such as visibly pushdown automata. Support for other structures like EFMSs and its variant like register automata (RA) will be incorporated in future updates.

AutomataLib is mainly composed of three parts, namely implementation, abstraction, and algorithms, as shown in Fig. 6. The implementation part contains generic implementations of DFAs, Mealy machines, and adapter modules. The abstraction layer consists of Java interfaces for the representation of various automata types. The algorithm part consists of algorithms for minimization, equivalence, or visualization.

4.2 Learnlib

LearnLib [133] library has been developed at the Dortmund University of Technology, Germany. The library is a free, open source, and implemented in Java for automata learning algo-

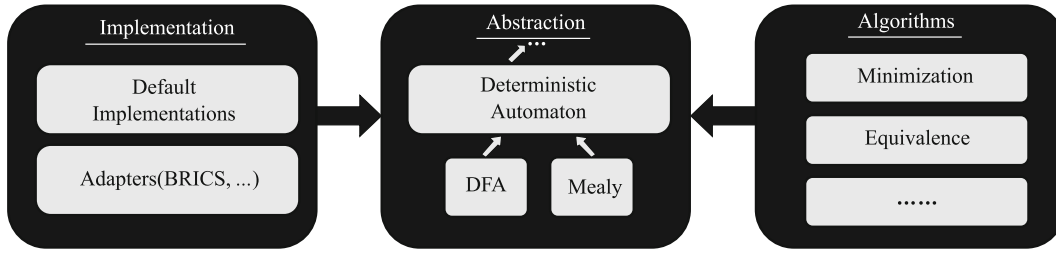


Fig. 6 Architecture of AutomataLib [132]

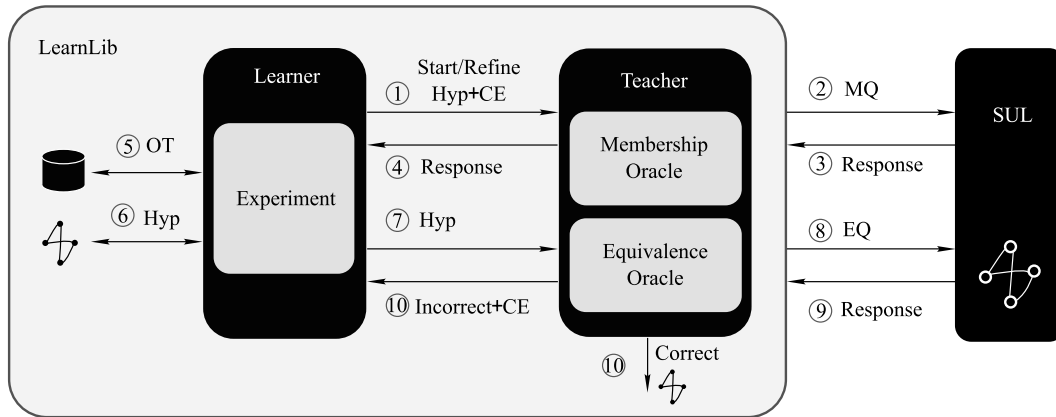


Fig. 7 Overview of learning model with LearnLib

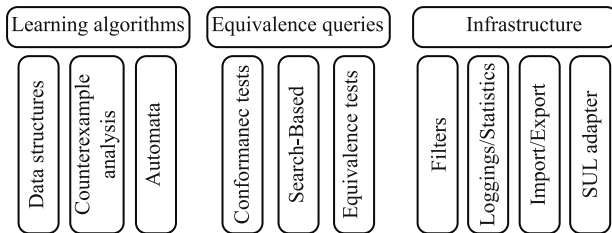


Fig. 8 An overview of basic LearnLib components

rithms. The sole objective of LearnLib is the provision of a framework for conducting research work on automata learning algorithms and their applications in real life. Besides equivalence checking algorithms, it also provides various active and passive learning algorithms targeting different modeling formalisms. Its modular and generic architecture allows us to extend it easily. In recent years, LearnLib has been used successfully to learn the behavioral models of diverse realistic systems, as shown in Section 5. Figure 7 shows the schematic overview of learning a model with the LearnLib library.

LearnLib consists of three main modules, which are automata learning module, infrastructure module, and equivalence queries module, as shown in Fig. 8. This figure has been adapted from the figure on the Learnlib web-page.

- 1) **Automata learning module:** It is the main module of LearnLib. It consists of different learning algorithms and their supported modeling structures. It also provides algorithms for handling data structures efficiently, which enable learning techniques to learn large-scale systems [132]. Besides, as the way of processing counterexamples has a significant impact on the learning complexity, so

Learnlib has been equipped with efficient counterexample handlers. Table 1 presents these algorithms along with other key features.

- 2) **Infrastructure module:** This module provides the infrastructure for query optimization, utilities for statistical parameters (e.g., no of MQs, no of EQs, memory consumption, running time, etc.), logging facility, a mechanism for storing and loading of hypotheses, and facility of SUL adapter. One of the goals of this module is to improve learning complexity by reducing MQs. To achieve this, it provides a parallelization component to divide queries among multiple workers, and various filters to eliminate duplicated queries.
- 3) **Equivalence queries module:** This module depends upon conformance testing, equivalence testing and various search-based techniques [132, 134, 135]. The process of finding a counterexample can be seen as an *equivalence query*. For a known target system, “perfect” equivalence queries are possible, and LearnLib uses Hopcroft and Karp’s algorithm [136] implemented in AutomataLib. In the case of the black-box system, equivalence queries can be approximated and answered by testing the target system. AutomataLib provides implementations of conformance algorithms that can find missing states, for example, W-method [95], Wp-Method [96], W-method randomized, Wp-Method randomized, random words, random walk, and complete depth bounded exploration.

4.3 LearnLib-based developed tools

LearnLib has been used in the development of tools like PSYCO [137, 138], Tomte [104] and ALEX [139]. In white-

box scenarios (where source code is accessible), the *Psyco* and *Sigma** [140] tools combine the *L** algorithm with symbolic execution to infer behavior models. The *Tomte* tool, which has been developed at the Radboud University of Nijmegen, utilizes learning algorithms provided by *LearnLib* for learning richer classes of models (e.g., EFSMs). *Tomte* fully automatically constructs abstractions for automata learning. The *ALEX* (automata learning experience) [139] is an easy to use tool which is developed in JAVA to infer automaton models of web applications and JSON-based web services.

4.4 The Libalf

The *Libalf* [141] is an “automaton learning framework” that has been developed at RWTH Aachen. It is free, open-source, and a comprehensive library for learning finite-state automata [141]. *Libalf* implements some well-known learning techniques such as Angluin’s *L**, RPNI, and Biermann’s learning approach. *Libalf* is highly flexible, modular in design, and learning algorithms for richer modeling formalisms like timed automata, Buchi automata, or probabilistic automata can also be incorporated into it. The supported algorithms, along with corresponding target models, are summarized in Table 1.

4.5 Automata-identification engine (AIDE)

AIDE is under the process of development. It has been implemented in *C#.Net*, and freely available on “CodePlex” hosting site. It is an open source tool for learning different kinds of automata, including DFA, deterministic Mealy machines, NFA, and Interface automata (IA). It supports active and passive learning approaches. To approximate equivalence oracle for dealing black-box systems, it is equipped with a *Testing Engine/testing tool*. Moreover, *AIDE* also provides the facility of *Automata tool* for generating different automata as benchmarks to test the tool and to evaluate different algorithms.

4.6 RALib

RALib [130] is an open source library and has been implemented in JAVA for active learning of register automata (a form of extended finite state machines). *RALib* is an extension of *LearnLib* [142] for automata learning and licensed under the *Apache License, Version 2.0*. It provides an extensible implementation to learn models of SULs in the form of register automata in an active way. Besides, it provides modules for output, mixing different tests on data values, typed parameters, and directly inferring models of JAVA classes. *RALib* also provides heuristics for finding counterexamples as well as a range of performance optimizations (e.g., reducing the length of counterexamples). Cassel et al. [143] evaluated *RALib* on a set of benchmarks and compared it with other tools like *Tomte* [104, 109] and *LearnLib^{RA}* (this version has functionality for learning Mealy machines and EFSMs) [62, 63] for learning EFSMs and showed that *RALib* is superior with respect to expressivity, features, and performance. We refer the interested readers to [143] for further details about the comparison between *RALib* and other tools that infer register automata. And we refer to [122] for viewing the details about the comparison between *Tomte* and *LearnLib^{RA}*. Results proved the fact that *LearnLib^{RA}* outperformed *Tomte*, particularly on the smaller models, but on the

other hand, *Tomte* required fewer tests to infer a model completely of larger models (SULs). It provides two tools that can be used from the shell: (i) an IO simulator and (ii) a Java class analyzer. The IO simulator uses a register automaton model as a SUL and can be used to evaluate different algorithms. The class analyzer can be used to infer models of java classes.

4.7 RALT

RALT [21] (Rich Automata Learning and Testing Tool) is a reverse-engineering tool, developed in JAVA, to infer the models of real systems taken as a black-box. It conjectures the behavior of a SUL in the form of a finite state machine by generating tests. It outputs finite-state models of a SUL in two formats: DOT and JFLAP. The resulted model is then viewed and analyzed by a visualizer. Graphviz, for example, is used for visualization of DOT formats. *RALT* worked under two assumptions. On the one hand, the SUL should be a finite state machine and must behave like a Mealy machine. This means that there is an output for every input. On the other hand, *RALT* requires a test driver for running tests on SUL and provides the results of tests back to *RALT*.

4.8 Symbolic automata library

Symbolic Automata Library [71, 131] is an efficient library that gives the provision of representing large (or infinite) alphabets concisely. It supports finite symbolic automata (SFA), symbolic finite transducer (SFT), and symbolic visibly pushdown automata (SVPA) with corresponding algorithms. Moreover, it also supports symbolic streaming string transducers and character theory of intervals. A short list of tools that are based on symbolic automata and transducers include Microsoft automata library, Bek, Bex, Fast, and Mona. These tools are available on a community site “rise4fun” that provides a web front end for running software engineering tools from a web browser.

4.9 Comparison of learning libraries

Various learning libraries/tools are being used in industry to infer models of realistic systems. A comparison of some important libraries is presented in Table 2.

5 Applications of model learning

Model learning is being applied successfully in diverse areas, including specification generation, software reverse engineering, MBT testing, formal verification, compositional reasoning, etc. The broad range and successful stories highlight the fact that active automata learning (model learning) is very promising and has become one of the best tools of formal methods. In the following subsections, we discuss briefly the applications of model learning in different domains.

5.1 Model extraction and manual analysis

There are many applications where the extracted models of small and medium sizes have been analyzed manually to find bugs. However, for large and complex models, other analysis techniques like model checking (which is discussed in the next section) are used. In network and security protocols, bugs or vulnerabilities in specification or implementation may cause catastrophic loss and model learning or in general reverse engineering of protocols expose and mitigate such problems effec-

Table 2 Overview of the most important learning libraries/tools

Tool	LT ^a	LM ^b	LP ^c	Key Features	CS ^d	Ref.
LearnLib	Active and Passive support; For algorithms see Table 1	DFA,NFA, Mealy,VPDA	JAVA;All OS running jre	Filters, normalizers, visualization, statistics, Logging facilities and Complemented by AutomataLib library	Updated regularly, current version is 0.14.0 (Feb, 2019)	[107]
LibAlf	Active and Passive support; For algorithms see Table 1	DFA, NFA, Mealy, visibly 1-counter automaton	C++;MS-Windows, Linux and Mac OS	Filters, statistics, visualization, normalizers and Logging facilities; Also, Complemented by two additional libraries: (i)liblangen (ii)AMoRE++	No updation, current version is 0.3, last updation was made on April, 2011.	[141]
AIDE	Active+Passive; (see Table 1)	DFA, NFA, Mealy, IA	C#.Net;MS windows	Support for: (i)Automata tool (ii) Testing tool	No updation	[144]
RALib	Active Learning; Algo: SL* [130]	EFSMs/RA	JAVA; All OS running jre	Support for IO simulator, Java class analyzer,multiple theories (testing equalities and inequalities with constants and fresh data values)	Updated regularly, Current version is 2.0 (March, 2019)	[143]
RALT	Active Learning; visit Table 1	Algos: DFA,Mealy, PFSM, K-Tree	JAVA;All OS running jre	Not public; Support for visualization, statistics, logging	Updated regularly,V 5.3.3 (Dec,2017)	[56]
TOMTE	Active Learning	ESFM/RA	JAVA;All OS running jre	Construction of abstraction fully automatically; Support for visualization, statistics, logging;	Updated regularly, Tomte-0.41 (Sep,2016)	[109]
ALEX	Active Learning	Mealy machine	JAVA ; All OS having Java JRE 8	Active learning of web applications, JSON-based REST services.Support for visualization,statistics,logging;	Current version is 1.6.1 (Dec, 2018)	[139]
PSYCO	Active; Combines dynamic and static analysis techniques	Symbolic behavioral interfaces	JAVA;All OS having Java JRE	Dependencies on jpf-core and jpf-jdart; Support for visualization, statistics	Last updation on Oct, 2016	[137]
Symbolic Automata Lib	Active Learning	SFA, SFT and SVPA	JAVA;All OS having Java JRE	Support for: Symbolic streaming string transducers, character theory of intervals, Microsoft automata library, Bek, Bex, Fast and Mona	Updation on regular basis	[131]

^a Learning type and supported algorithms.^b Learning models (target models).^c Implementation language and supported platform.^d Current status of tool.

tively. Aarts et al. [145] used model learning and abstraction to learn the model of the biometric passport that describes how the passport behaves to certain input sequences. In a similar kind of work, Aarts et al. [146] inferred the state machine models of smart banking cards using model learning and abstraction techniques. Smeenk et al. [147] applied model learning to infer the model of the control software of the printer (embedded control software). Cho et al. [101] applied model learning techniques on botnet command and control protocols for inferring models by placing an emulator/mapper component between learning software and botnet servers. Aarts et al. [99, 100] proposed a framework that modifies the learning technique so that it can have data parameters in messages and states for the generation of components with large message alphabets. To infer the models of SIP protocol, they used the concept of predicate abstraction and interface LearnLib with ns-2 protocol simulator that contains the SIP implementation. In a similar nature of works, models of TCP protocols were inferred from TCP implementations [148, 149].

5.2 Combining model learning and model checking

Peled et al. [30] was the first one who introduced the idea of combining model learning and model checking with the name *black-box checking*. Although the learned models can be analyzed manually [145, 146] but it is tedious and time consuming since the inferred models may be too complex for manual inspection.

In one approach, model checking was used for the analysis of models that were made manually by using protocol standards. The employed approach captures some bugs, but this activity is time-consuming and error-prone. Moreover, protocol implementations often do not exist in accordance with specifications. Detection of implementation-specific bugs or vulnerabilities becomes difficult by using model checking in this style. Fiterău-Broștean et al. [149] learned the models of TCP protocols and highlight that TCP implementations in Windows 8 and Ubuntu 13.10 violate the standard (RFC 793) specifications. De Ruiter and Poll [28] interred and analyzed three implementations of TLS protocols and found security flaws due to violations of the standard. Chalupar et al. [102] configured Lego robot and used model learning technique to reverse-engineering the implementations embedded into hand-held smart card readers, which are used in internet banking. The analysis of the inferred models reveals that implementations violate the standard. Tijssen [150] also reported the violation of standard in the implementation of SSH protocol.

To analyze TCP implementations, Fiter et al. [151] combine both techniques, i.e., model learning and model checking, to obtain formal models of Linux, Windows, and FreeBSD TCP server & client implementations. They analyzed the learned models with the nuSMV model checker to check the results when these components interact (e.g., a FreeBSD client and a Linux server) with each other. In another study, to analyze the implementations of SSH protocol, Fiterău-Broștean et al. [152]

used model learning in combination with abstraction techniques for inferring state machines models of Bitwise, OpenSSH, and DropBear SSH server. Later on, they applied model checking technology on learned models for further analysis. They selected many security and functional properties from SSH RFC specifications and formalized them in LTL language. The authors then checked the formalized properties on the obtained models of SSH protocol using the nuSMV model checker and found several standard violations.

In fact, all the violations of standard's specifications reported above have been discovered by the application of model-checking technologies to the learned models using model learning techniques.

5.3 Re-engineering of software modules

Learned models can be utilized to compare different implementations or revisions of a system. Schuts et al. [153] presented an approach at Philips and used model learning in combination with equivalence checking technologies to re-engineer an implementation of the legacy control component. They applied model learning to old implementation and at the same time, on the new implementation. In their study, they used the equivalence model checker for the comparison of learned models. Bainczyk et al. [139] presented a tool called *ALEX* to infer the models of web application fully automatically. The beauty of *ALEX* is that it supports mixed-mode learning, i.e., REST and web services can be run simultaneously in a learning experiment. This feature is excellent when we compare front-end and back-end functionalities of a web application. Neubauer et al. presented an Active Continuous Quality Control (ACQC) system in which they use active automata learning on subsequent versions of web applications and then analyzed the learned models for spurious behavioral changes between versions [154, 155].

5.4 Model learning and testing techniques

The testing of black-box software components is a challenging task. One solution to handle this problem is combining testing and learning techniques so that the learned models of software components can be used to explore unknown implementation and ease the testing efforts. In recent years, profound progress has been made in this direction and receiving much attention to the testing community. Fuzz testing (or fuzzing) technique is being used successfully to discover implementation and security errors in software, networks, and operating systems. De Ruiter et al. [28] used a model learning technique to infer state machines from protocol implementations. They called this technique as protocol state fuzzing because it involves fuzzing different sequences of messages. The inferred state machines were then analyzed manually to look for spurious behavior that can be an indication of logical flaws in a program. In a similar kind of works, Verleg et al. [156] and Fiterău-Broștean et al. [152] used state fuzzing to infer the state models of SSH implementations. Hagerer et al. [26] presented an approach called *regular extrapolation* to infer the models of telecommunication systems for regression testing. Groce et al. [157] implemented a technique called *AMC* (adaptive model checking) for automatic verification, to handle the inconsistency between a system and its inferred model. Shu and Lee [22] proposed a learning-based

technique to infer the models of protocol implementation in the form of the symbolic parameterized extended finite state machine (SP-EFSM) for automatic testing of security properties.

The integration of prefabricated third-party components, which are loosely coupled in a distributed architecture, are being used extensively in designing complex systems like telecom services. Due to this, the system integrator faces many problems while integrating such black-box COTS components. Li et al. [158] addressed this problem and proposed a technique that combines model learning with test generation techniques. In another work, Li et al. [128] used the same methodology but focused on richer modeling formalisms that are more expressive for designing complex systems. They proposed a learning algorithm that was able to learn a black-box component in the form of an I/O parameterized model (called PFSM). Their sole objective was to facilitate integrators, which may derive systematic tests to analyze component interactions. In other research work, Shahbaz et al. [21, 159–161] analyzed PFSM models, to deal with the above testing issues in the environment where behavioral models of components and documentation are unavailable. Groz et al. [59] combined three techniques, i.e., inference, testing, and reachability, to study the verification of a modular system. They used reachability analysis on inferred models to find intermittent errors and compositional problems. Walkinshaw et al. [162] addressed the challenging job of test generation to achieve the functional coverage in the environment where complete specification was missing. They highlighted the fact that inductive testing is more efficient than classical black-box techniques for test generation.

5.5 Compositional reasoning/verification

Model learning is also contributing a lot to the promising domain of compositional reasoning and verification. In compositional reasoning, each software component (like a library function or a concurrent thread) is treated in isolation without any knowledge of software contexts like the rest part of the software or any other environment thread where it will be installed.

Compositional verification presents an efficient way of handling the state explosion problem associated with model checking. In compositional verification, by following the approach of “divide and conquer,” the properties of the whole system are broken down into the properties of its components. In this way, if all components satisfy their corresponding properties, it means that the whole system is satisfied. However, components that are being model checked in separation may satisfy properties in a specific context/environment. It generates the requirement for the assume-guarantee style of reasoning [163, 164]. It is a compositional technique for improving the scalability of model checking

Several theoretical frameworks exist for assume-guarantee reasoning, but they are less practical due to the involvement of non-trivial human interaction. Cobleigh et al. [163] presented a framework for performing assume-guarantee reasoning in an incremental and fully automated way. To analyze a component against a property their approach produced assumptions using the L^* algorithm in combination with model checking (for counterexamples) that is the requirement of the environment for satisfying the property to hold. If the property holds, the process

will return “true” with a guarantee of termination, and if not successful, then a counterexample will be returned. They implemented their technique with the LTSA tool and applied successfully to some systems in NASA. Model learning has been employed in another work performed by He et al. [165], where the authors presented a learning-based assume-guarantee regression verification technique. In model checking, much internal information is computed during the running process of verification. Also, the two consecutive revisions have some common behavior which may be utilized in the verification process. For example, when one revision is completed, then the internal information computed by model checking may still be useful for the verification of the next revision. He et al. [165] fully utilized this idea in their technique, and they reused the contextual assumptions of the previous round of verification in the current round. Similarly, there are some other case studies where model learning algorithms have been utilized for learning assumptions for compositional verification automatically [166–168].

5.6 Other diverse applications

Apart from the above mentioned applications of model learning there are numerous other cases where it has been used successfully. The range includes, building of compositional synthesis algorithm [169] based on learning-based assume-guarantee, detection of malware [108], the construction of reactive controllers to solve safety games [170], the setting of CTI systems [171, 172], testing brake-by-wire system [173], minimizing partially specified systems [174], GUI testing [175], typestate analysis [176, 177], and learning Java programs [9].

5.7 Comparison of model learning applications

A comparison of some important applications of model learning in various domains is presented in Table 3.

6 Achievements, challenges and future work

The overarching goal of model learning is to provide learning techniques for the construction of accurate and reliable models which are utilized in formal validation techniques. In the following subsections, we discuss the recent advancements, challenges and future goals in the domain of model learning.

6.1 Achievements

During recent years, tremendous progress has been made in the field of model learning, and Table 3 presents its successful applications in multidisciplinary areas of real life. The significant progress made in learning & equivalence checking algorithms, model inferring formalisms, advancements in learning techniques, and learning tools have played a crucial role in scaling the applications of model learning to realistic systems.

6.1.1 Advances in algorithms development

Modeling formalisms: Basic learning algorithms like L^* are successful only for small state machines. For realistic systems, we have to generalize existing learning algorithms to richer modeling formalisms like EFSMs. These models have more expressive power as compared to simple state machines and have the ability to store, process, and communicate data values. Howar et al. [62] extended active model learning to register automata and demonstrated their approach by learning models of semantic interfaces of data structures. Register automata have

the expressive power to model data and control flows between data parameters of inputs, outputs, and registers. Aarts et al. [182] also presented a novel approach for inferring register automata models, which can compare the values of registers and data parameters for equality. They implemented their approach in the Tomte tool, which utilizes the CEGAR technique for the construction of mapper automatically. Isberner [27] developed a learning algorithm for inferring visibly push-down automata. As reactive systems can be modeled easily as Kripke structures, Meinke and Sindhu [183] presented a model learning algorithm IKL for such structures. Volpato and Tretmans [184] presented an algorithm for learning the models in the form of LTS for non-deterministic and input-enabled systems.

Efficiency: Angluin [39] introduced the L^* algorithm to learn models in the form of DFA, which was later extended by Niese [26] to model Mealy machines. Kearns and Vazirani [88] improved L^* algorithm by replacing the observation table with discrimination tree which reduces the number of MQs. As for finding counterexamples, the learner, i.e., L^* adds all the prefixes of a counterexample in rows of the table and again starts posing MQs to refine the hypothesis. Counterexample generator may produce a long, not minimal counterexamples which result in posing of numerous redundant MQs. Rivest and Schapire [50] improved the L^* algorithm by adding an appropriate suffix as a column instead of adding rows. Isberner et al. [89] proposed TTT algorithm that is considered to be more efficient. The differentiating characteristic of this algorithm is related to the organization of observations which is redundancy free and can be used to achieve optimal (linear) space complexity. TTT algorithm stores data in data structures of tree-shaped which leads to a compact representation. It is well suited in the context of runtime verification where counterexamples may be very long.

Shahbaz and Groz [55] worked to improve existing learning algorithm in terms of reducing the worst-time learning complexity. Groz et al. [185] made improvement in L^* algorithm targeting parameterized inputs, input sets and counterexamples handling. To speed up the learning process, some techniques have been developed including parallelization and checkpointing proposed by Henrix [113]. Learning tool like LearnLib is single-threaded, which means that if we would have access to a large computer cluster, then the required execution time (or learning time) will remain the same. The total learning time can be minimized by running multiple instances of the SUL at the same time. Hence, the total amount of work done remains unchanged, but by distributing it (parallelization), the work can be done in less time (reduced learning time). Learning time has also been improved by checkpointing, i.e., by saving and restoring software states.

6.1.2 Advances in tools and libraries

We need robust and versatile implementations of learning algorithms along with surrounding infrastructure for the application of model learning in practical life. In this regard, a great amount of work in the form of learning tools and libraries have been done. Most of these libraries are free, open source, and implemented in JAVA. These libraries provide a sophisticated set of algorithms both for active and passive learning for various

Table 3 Comparison of model learning applications

Application	Learned model	Library/Tool	Ref.
(Model extraction and manual analysis)			
Inferring model of biometric passport	Mealy FSM	LearnLib	[145]
Inferring model of banking smart cards	Mealy FSM	LearnLib	[146]
To infer the model of control software of printer (embedded control software)	Mealy FSM	LearnLib	[147]
To analyze botnet command and control protocols	FSM (protocol state machine)	Use of learning algo	[101]
Inferring model of SIP protocol to verify its conformance with RFC specifications	FSM	LearnLib ns-2, Tomte	[99, 100]
To infer the models of TCP implementations with automatic abstraction notion	FSM	Tomte	[109]
(Combining model learning and model checking)			
Analysis of TCP protocols in Windows 8 and Ubuntu 13.10 implementations	FSM	LearnLib	[148, 149]
Combining model learning and model checking to analyze TCP implementations	LearnLib	nuSMV	[151]
TLS Protocol (Protocol State Fuzzing)	FSM	LearnLib	[28]
Formal Verification of SSH Security Protocol	FSM	LearnLib	[150]
To analyze different SSH implementations	LearnLib	nuSMV	[152]
Reverse engineering of smartcard readers(e.dentifier2) using Lego robot	FSM	LearnLib	[102]
Application of model learning in legacy software component	FSM	LearnLib,mCRL2	[153]
(Model learning and testing techniques)			
Testing (regression testing) of telecommunication systems using regular extrapolation	FSM	Adapted L*	[26]
To address the problem of integration testing of COTS using learning based testing	PFSM	Adapted L* learning algo	[21, 128, 159–161]
Using automata learning to systematically test black box systems(dynamic testing)	FSM	LearnLib	[178]
Learning based testing: To test security properties of protocol implementations (N-S-L,TMN,SSL 3.0)	SP-EFSM	Adapted L* algo	[22]
Combining inference, testing, and reachability techniques for testing of modular system	IOTS (FSM with multiple outputs)	Customized learner for IOTS	[59]
Application of inductive testing to realistic black-box systems (learning based test generation)	FSM	customized learning algo	[162]
Protocol state fuzz testing of SSH security protocols	Mealy FSM	LearnLib	[28, 152, 156]
(Re-engineering of software modules)			
Re-engineering an implementation of the legacy control component	Mealy FSM	LearnLib	[153]
To compare front and back ends functionalities of a web application	FSM	ALEX	[139]
Comparing different versions of models in Active Continuous Quality Control (ACQC) system	Mealy machines and kripke structures	LearnLib	[154, 155]
(Compositional reasoning/verification)			
Using model learning to learn assumptions for compositional verification	DFA	Adapted L* algo	[163–168]
(Other diverse applications of model learning)			
Software and compositional model checking(Program verification)	State machine model	Customized learning algo	[61, 179]
To apply model learning in malware detection	FSM	Adapted L*	[108]
Guided GUI testing of Android Apps	Extended deterministic labeled transition systems (ELTS)	SwiftHand	[175]
Learning of industrial control software	FSM	LearnLib	[147, 180]
To models of stateful tpestates	FSM	TzuYu tool	[176, 177]
Model-Based testing of IoT communication using model learning	FSM	Use of a learning algo	[181]
Learning interface specifications for Java classes	State machine model	Adapted L*, model checker	[176]
Learning and finding difference of similar programs	SFA	SFADiff	[73]
Regex (regular expression) processing	SFA	SFA Learner	[72]

automata models, along with a variety of equivalence checking algorithms. We referred the interested readers to Section 4 of this paper for more details.

6.2 Current challenges and future work

No doubt, significant progress has been made in the domain of active model learning, as highlighted in the last subsection 1, but the area is still in the developing phase and getting maturity with time. There are still many limitations and problems that need to be solved to bring it from the current academic level to practical life.

6.2.1 Extending abstraction techniques

We have discussed in Section 4 that to learn models of realistic systems, which have large inputs and outputs, *abstraction* plays a key role. Aarts et al. [99, 100] proposed a framework that modifies the learning technique so that it can have data parameters in messages and states for the generation of components with large message alphabets. The current versions of learning algorithms can model systems that only test the equality of data parameters, but operations on data parameters are not allowed. These algorithms model the SUL in the form of RA, which is a big breakthrough in model learning, but its scope is still limited due to the restrictions imposed on operations on data. Therefore, more work is required to extend the current abstraction/mapper techniques, which allow operations on data as well as comparisons (like, “less than” or “greater than”) in guard statements to model realistic systems. To achieve this, we may also benefit from existing research work. For example, Ernst et al. [186] developed *Daikon* tool, which is an implementation of dynamic detection of likely invariants; that is, the Daikon invariant detector reports likely program invariants. It is able to highlight relationships (“less than” or “greater than”) between variables.

6.2.2 Improving testing techniques

Predicting the correctness of inferred models is another challenge that model learning still has to be dealt with. As model learning technique produces models by posing tests that are finite in number, so we can never be sure about the reliability of the learned model. Using an upper bound on the number of states of a SUL, one may employ the traditional testing technique, i.e., conformance testing that provides a test suite assuring the correctness of the learned model. However, this testing strategy is not feasible because the required number of test queries grows exponentially with the number of states of a SUL. Aarts [109] concluded in his research work that test selection and coverage are still a big barrier in the development and application of active learning tools. Therefore, more research work on enhancing testing techniques for equivalence approximation, especially for non-deterministic systems, is required. The objective is to measure and quantify the correctness of the hypothesized model to assess its quality.

To achieve the aim, we can also utilize existing research work to improve testing techniques for predicting the quality of learned models. For example, Bos et al. [93] studied a general class of distance metrics for deterministic Mealy machines and developed a quality metric for inferred models. They introduced a so-called “Comparator” that can be used to enforce that

the quality of the hypotheses (intermediate models) obtained during model learning always increases (w.r.t. to the introduced metric). Besides, Chen et al. [94] introduced a novel technique for verification and model synthesis of sequential programs. Their learning framework for program verification is based on the framework of probably approximately correct the (PAC) learning. The use of PAC result allows them to quantify the confidence in the verification result in the absence of a perfect equivalence oracle.

6.2.3 Learning non-deterministic behavior

In practice, we often encounter systems that are deterministic but sometimes may also exhibit non-deterministic behavior. Aarts et al. [145] applied a model learning technique to infer a model of a biometric passport. According to the specification of the passport, its implementation must be deterministic, but the authors observed non-deterministic behavior as well during the learning process. Therefore, extending current model learning approaches to handle the non-deterministic behavior of systems effectively is a potential future work. For this purpose, we may employ algorithms based on game theory [187] to drive non-deterministic systems to certain states for the checking of non-determinism by executing the same input again and again (from these certain states). We can also benefit from existing research work on learning of non-deterministic systems [78, 79, 184, 188–190].

6.2.4 Combining model inference techniques

Combining the power of learning techniques can be propitious for many applications. However, their integration for the selection of the best mixture of techniques, to obtain a behavior model for a specific system, is a challenging job. For this, more work is required, which can benefit from existing efforts made to combine model inference techniques. For instance, the combination of active and passive learning approaches [191, 192] can be useful by first constructing a model using a passive learning technique and then refining it with the help of active learning technique or by using the passively learned model as an additional oracle during active learning. Some other techniques that can also be beneficial for future work in this regard include incremental sequential learning technique for the Mealy model [193], model learning using homing sequence [50] which is useful in situations where the system under testing cannot be “reset” and checking of black-box systems [194].

6.2.5 Learning of more expressive models

The limited expressive power of prevailing modeling formalisms (e.g., a Mealy machine) used by model learning techniques is also a challenge to model realistic complex systems. For instance, a Mealy machine is a good option for modeling reactive systems, but the realistic system may have zero or more outputs corresponding to one input. It may also be possible that the behavior of the system is time-dependent, e.g., the output may only occur if the corresponding input is applied for a certain amount of time, etc. The Mealy machine model does not handle such behavior in realistic systems due to its low power of expressiveness. For example, Fiter et al. [151] eliminated timing-based behavior to squeeze TCP implementation into Mealy machine. Future work can utilize the existing ef-

forts made to learn real-world systems in more expressive models (e.g., EFSM models). Register automata, for example, is a particular extension for which model learning algorithms have been developed [195]. In literature, there exist two approaches to learn models in the form of register automata. First one is developed by Cassel et al. [66] and has been implemented in the learning tools LearnLib [133] and RALib [143]. Aarts et al. [182] has developed the second approach to learn the register automata and has been implemented in the software tool Tomte [109].

6.2.6 Improving model learning tools

Besides efficient learning algorithms, the surrounding infrastructure for rapid assembling of learning setups is also compulsory. To satisfy these needs, different learning tools have been developed. However, still more efforts are required to make these tools (for example, LearnLib, libalf, RALib, RALT or Tomte, etc., as shown in Table 2) able to learn behaviors involving the notions of timing constraints, non-determinism, operations on data along with comparisons, handling of large (or infinite) alphabets, etc.

7 Conclusion

The quality of software systems, especially the correct functioning of safety and mission-critical systems, is of great concern and demands effective approaches that complement the existing testing and verification techniques by addressing their shortcomings. In this regard, we have surveyed an emerging technique called *model learning*, which is highly effective in exploring the hidden structures of black-box systems thoroughly. We have focused on and reviewed the active model learning framework, techniques, algorithms, tools, and applications in multidisciplinary domains. Table 1 surveys some well-known learning algorithms, their supported modeling formalisms, and key features for inferring models of realistic systems. To ease the learning process for realistic systems, Table 2 reviews different model learning libraries and tools.

No doubt, the model learning technique is promising for real-life systems, which is demonstrated in Table 3, but the field is still in the developing phase and facing some challenges. To bring this technique to a level where it can be readily applied to real systems, research work regarding modeling formalisms for more expressive models, algorithms development for learning and validation, efficient abstraction techniques, and feature-rich learning libraries is required. From the analysis of our comparison summaries, we conclude that model learning is efficient, emerging, and promising technique, and can complement existing testing and verification techniques by providing learned models of black-box systems fully automatically.

Acknowledgements We would like to gratitude Mr. Naeem Irfan, Mr. Kashif Saghar, and Mr. Markus Frohme TU Dortmund for valuable discussions on Model learning. This work was supported by the National Natural Science Foundation of China (NSFC) (Grant Nos. 61872016, 61932007 and 61972013).

References

1. Blair M, Obenski S, Bridickas P. Patriot missile defense: software problem led to system failure at dhahran. Report GAO/IMTEC-92-26, 1992
2. Lions J L. Ariane 5 flight 501 failure. Report by the Inquiry Board, 1996

3. Stephenson A G, Mulville D R, Bauer F H, Dukeman G A, Norvig P, LaPiana L S, Rutledge P J, Folta D, Sackheim R. Mars Climate Orbiter Mishap Investigation Board Phase I Report. NASA, Washington, DC, 1999
4. Leveson N G, Turner C S. An investigation of the therac-25 accidents. Journal of Computer, 1993, 26(7): 18–41
5. Coe T. Inside the pentium FDIV bug. DR Dobbs Journal, 1995, 20(4): 129
6. Ball T, Rajamani S K. The slam project: debugging system software via static analysis. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2002, 1–3
7. Henzinger T A, Jhala R, Majumdar R, Sutre G. Lazy abstraction. Journal of ACM SIGPLAN Notices, 2002, 37(1): 58–70
8. Walkinshaw N, Bogdanov K, Ali S, Holcombe M. Automated discovery of state transitions and their functions in source code. Journal of Software Testing, Verification and Reliability, 2008, 18(2): 99–121
9. Walkinshaw N, Bogdanov K, Holcombe M, Salahuddin S. Reverse engineering state machines by interactive grammar inference. In: Proceedings of Working Conference on Reverse Engineering. 2007, 209–218
10. Biermann A W, Krishnaswamy R. Constructing programs from example computations. Journal of IEEE Transactions on Software Engineering, 1976, 3: 141–153
11. Muller-Olm M, Schmidt D A, Steffen B. Model-checking: a tutorial introduction. In: Proceedings of International Static Analysis Symposium. 1999, 330–354
12. Clarke E M, Grumberg O, Peled D. Model Checking. MIT Press, 1999
13. Baier C, Katoen J P. Principles of Model Checking. MIT Press, 2008
14. Broy M, Jonsson B, Katoen J P, Leucker M, Pretschner A. Model-based testing of reactive systems. Lecture Notes in Computer Science, 2005
15. Utting M, Legeard B. Practical Model-based Testing: a Tools Approach. Elsevier, 2010
16. Arbab F. Reo: a channel-based coordination model for component composition. Journal of Mathematical Structures in Computer Science, 2004, 14(3): 329–366
17. Ball T, Bounimova E, Cook B, Levin V, Lichtenberg J, McGarvey C, Ondrusek B, Rajamani S K, Ustuner A. Thorough static analysis of device drivers. Journal of ACM SIGOPS Operating Systems Review, 2006, 40(4): 73–85
18. Hungar H, Niese O, Steffen B. Domain-specific optimization in automata learning. In: Proceedings of International Conference on Computer Aided Verification. 2003, 315–327
19. Margaria T, Niese O, Raffelt H, Steffen B. Efficient test-based model generation for legacy reactive systems. In: Proceedings of the 9th IEEE International High-Level Design Validation and Test Workshop. 2004, 95–100
20. Muccini H, Polini A, Ricci F, Bertolino A. Monitoring architectural properties in dynamic component-based systems. In: Proceedings of International Symposium on Component-Based Software Engineering. 2007, 124–139
21. Shahbaz M. Reverse engineering enhanced state models of black box software components to support integration testing. Doctoral Dissertation, 2008
22. Shu G, Lee D. Testing security properties of protocol implementations-a machine learning based approach. In: Proceedings of the 27th International Conference on Distributed Computing Systems. 2007
23. Aarts F, Vaandrager F. Learning I/O automata. In: Proceedings of International Conference on Concurrency Theory. 2010, 71–85
24. Moore E F. Gedanken-experiments on sequential machines. Journal of Automata Studies, 1956, 34: 129–153
25. Berg T, Grinchtein O, Jonsson B, Leucker M, Raffelt H, Steffen B. On the correspondence between conformance testing and regular inference. In: Proceedings of International Conference on Fundamental Approaches to Software Engineering. 2005, 175–189

26. Hagerer A, Hungar H, Niese O, Steffen B. Model generation by moderated regular extrapolation. In: *Proceedings of International Conference on Fundamental Approaches to Software Engineering*. 2002, 80–95
27. Isberner M. Foundations of active automata learning: an algorithmic perspective. Doctoral Dissertation, 2015
28. De Ruiter J, Poll E. Protocol state fuzzing of TLS implementations. In: *Proceedings of the 24th USENIX Security Symposium*. 2015, 193–206
29. Vaandrager F. Model learning. *Journal of Communications of ACM*, 2017, 60(2): 86–95
30. Peled D, Vardi M Y, Yannakakis M. Black box checking. *Journal of Automata, Languages and Combinatorics*, 2002, 7(2): 225–246
31. Ammons G, Bodík R, Larus J R. Mining specifications. *Journal of ACM Sigplan Notices*, 2002, 37(1): 4–16
32. Lo D, Khoo S C. Smartic: towards building an accurate, robust and scalable specification miner. In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2006, 265–275
33. Lorenzoli D, Mariani L, Pezzè M. Automatic generation of software behavioral models. In: *Proceedings of the 30th International Conference on Software Engineering*. 2008, 501–510
34. Bennaceur A, Meinke K. Machine learning for software analysis: models, methods, and applications. In: Bennaceur A, Hähnle R, Meinke K, eds. *Machine Learning for Dynamic Software Analysis: Potentials and Limits*. Springer, Cham, 2018
35. Starkie B, Zaanen V M, Estival D. The tenjinno machine translation competition. In: *Proceedings of International Colloquium on Grammatical Inference*. 2006, 214–226
36. Starkie B, Coste F, Zaanen V M. The omphalos context-free grammar learning competition. In: *Proceedings of International Colloquium on Grammatical Inference*. 2004, 16–27
37. Walkinshaw N, Lambeau B, Damas C, Bogdanov K, Dupont P. Stamina: a competition to encourage the development and assessment of software model inference techniques. *Journal of Empirical Software Engineering*, 2013, 18(4): 791–824
38. Combe D, De La Higuera C, Janodet J C. Zulu: an interactive learning competition. In: *Proceedings of International Workshop on Finite-State Methods and Natural Language Processing*. 2009, 139–146
39. Angluin D. Learning regular sets from queries and counterexamples. *Journal of Information and Computation*, 1987, 75(2): 87–106
40. Gold E M. Language identification in the limit. *Information and Control*, 1967, 10(5): 447–474
41. Valiant L G. A theory of the learnable. *Journal of Communications of ACM*, 1984, 27(11): 1134–1142
42. Angluin D, Smith C H. Inductive inference: theory and methods. *Journal of ACM Computing Surveys (CSUR)*, 1983, 15(3): 237–269
43. Gasarch W I, Smith C H. Learning via queries. *Journal of ACM (JACM)*, 1992, 39(3): 649–674
44. Watanabe O. A framework for polynomial-time query learnability. *Journal of Mathematical Systems Theory*, 1994, 27(3): 211–229
45. Blumer A, Ehrenfeucht A, Haussler D, Warmuth M K. Learnability and the vapnik-chervonenkis dimension. *Journal of ACM (JACM)*, 1989, 36(4): 929–965
46. Pitt L, Valiant L G. Computational limitations on learning from examples. *Journal of ACM (JACM)*, 1988, 35(4): 965–984
47. Pitt L, Warmuth M K. Prediction-preserving reducibility. *Journal of Computer and System Sciences*, 1990, 41(3): 430–467
48. Angluin D, Kharitonov M. When won't membership queries help? *Journal of Computer and System Sciences*, 1995, 50(2): 336–355
49. Steffen B, Howar F, Merten M. Introduction to active automata learning from a practical perspective. In: *Proceedings of International School on Formal Methods for the Design of Computer, Communication and Software Systems*. 2012, 256–296
50. Rivest R L, Schapire R E. Inference of finite automata using homing sequences. *Journal of Information and Computation*, 1993, 103(2): 299–347
51. Groz R, Simao A, Petrenko A, Oriat C. Inferring finite state machines without reset using state identification sequences. In: *Proceedings of International Conference on Testing Software and Systems*. 2015, 161–177
52. Groz S AR, Petrenko A, Oriat C. Inferring fsm models of systems without reset. In: Bennaceur A, Hähnle R, Meinke K, eds. *Machine Learning for Dynamic Software*. Springer, Cham, 2018
53. Hopcroft J E, Motwani R, Ullman J D. Introduction to automata theory, languages, and computation. *ACM Sigact News*, 2001, 32(1): 60–65
54. Niese O. An integrated approach to testing complex systems. Doctoral Dissertation, Technical University of Dortmund, Germany, 2003
55. Shahbaz M, Groz R. Inferring mealy machines. In: *Proceedings of International Symposium on Formal Methods*. 2009, 207–222
56. Shahbaz M, Li K, Groz R. Learning parameterized state machine model for integration testing. In: *Proceedings of Computer Software and Applications Conference*. 2007, 755–760
57. Aarts F, Kuppens H, Tretmans J, Vaandrager F, Verwer S. Improving active mealy machine learning for protocol conformance testing. *Journal of Machine Learning*, 2014, 96(1–2): 189–224
58. Walkinshaw N, Derrick J, Guo Q. Iterative refinement of reverse-engineered models by model-based testing. In: *Proceedings of International Symposium on Formal Methods*. 2009, 305–320
59. Groz R, Li K, Petrenko A, Shahbaz M. Modular system verification by inference, testing and reachability analysis. In: Suzuki K, Higashino T, Ulrich A, Hasegawa T, eds. *Testing of Software and Communicating Systems*. Springer, Berlin, Heidelberg, 2008
60. Huima A. Implementing conformiq qtronic. In: Petrenko A, Veanes M, Tretmans J, Griekamp W, eds. *Testing of Software and Communicating Systems*. Springer, Berlin, Heidelberg, 2007
61. Jhala R, Majumdar R. Software model checking. *Journal of ACM Computing Surveys (CSUR)*, 2009, 41(4): 21
62. Howar F, Steffen B, Jonsson B, Cassel S. Inferring canonical register automata. In: *Proceedings of International Workshop on Verification, Model Checking, and Abstract Interpretation*. 2012, 251–266
63. Howar F, Isberner M, Steffen B, Bauer O, Jonsson B. Inferring semantic interfaces of data structures. In: *Proceedings of International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. 2012, 554–571
64. Merten M, Howar F, Steffen B, Cassel S, Jonsson B. Demonstrating learning of register automata. In: *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2012, 466–471
65. Isberner M, Howar F, Steffen B. Learning register automata: from languages to program structures. *Journal of Machine Learning*, 2014, 96(1–2): 65–98
66. Cassel S, Howar F, Jonsson B, Steffen B. Active learning for extended finite state machines. *Journal of Formal Aspects of Computing*, 2016, 28(2): 233–263
67. Maler O, Mens I E. Learning regular languages over large alphabets. In: *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2014, 485–499
68. Argyros G, D'Antoni L. The learnability of symbolic automata. In: *Proceedings of International Conference on Computer Aided Verification*. 2018, 427–445
69. Drews S, D'Antoni L. Learning symbolic automata. In: *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2017, 173–189
70. Argyros G, Stais I, Kiayias A, Keromytis A D. Back in black: towards formal, black box analysis of sanitizers and filters. In: *Proceedings of IEEE Symposium on Security and Privacy*. 2016, 91–109
71. Veanes M, Hooimeijer P, Livshits B, Molnar D, Björner N. Symbolic

- finite state transducers: algorithms and applications. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2012, 137–150
72. Veanes M. Applications of symbolic finite automata. In: Proceedings of International Conference on Implementation and Application of Automata. 2013, 16–23
 73. Argyros G, Stais I, Jana S, Keromytis A D, Kiayias A. Sfadiff: automated evasion attacks and fingerprinting using black-box differential automata learning. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. 2016, 1690–1701
 74. Alur R, Dill D L. A theory of timed automata. *Journal of Theoretical Computer Science*, 1994, 126(2): 183–235
 75. Grinchtein O, Jonsson B, Leucker M. Learning of event-recording automata. *Journal of Theoretical Computer Science*, 2010, 411(47): 4029–4054
 76. Clark A, Thollard F. Pac-learnability of probabilistic deterministic finite state automata. *Journal of Machine Learning Research*, 2004, 5(May): 473–497
 77. Castro J, Gavaldà R. Towards feasible pac-learning of probabilistic deterministic finite automata. In: Proceedings of International Colloquium on Grammatical Inference. 2008, 163–174
 78. Yokomori T. Learning Non-Deterministic Finite Automata From Queries and Counterexamples. Oxford University Press, 1994
 79. Denis F, Lemay A, Terlutte A. Learning regular languages using non deterministic finite automata. In: Proceedings of International Colloquium on Grammatical Inference. 2000, 39–50
 80. Alur R, Courcoubetis C, Henzinger T A, Ho P H. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In: Grossman R L, Herode A, Ravn A P, eds. Hybrid Systems. Springer, Berlin, Heidelberg, 1993
 81. Van Der Aalst W, Adriansyah A, De Medeiros A K A, Arcieri F, Baier T, Blickle T, Bose J C, Brand V. d P, Brandtjen R, Buijs J, et al. Process mining manifesto. In: Proceedings of International Conference on Business Process Management. 2011, 169–194
 82. de la Higuera C, Janodet J C. Inference of ω -languages from prefixes. *Theoretical Computer Science*, 2004, 313(2): 295–312
 83. Hopcroft J E, Motwani R, Ullman J D. Introduction to automata theory, languages, and computation. *Journal of ACM Sigact News*, 2001, 32(1): 60–65
 84. Bollig B, Habermehl P, Kern C, Leucker M. Angluin-style learning of NFA. In: Proceedings of International Joint Conference on Artificial Intelligence. 2009, 1004–1009
 85. Merten M, Howar F, Steffen B, Margaria T. Automata learning with on-the-fly direct hypothesis construction. In: Leveraging Applications of Formal Methods, Verification, and Validation. Springer, 2012
 86. Maler O, Pnueli A. On the learnability of infinitary regular sets. *Journal of Information and Computation*, 1995, 118(2): 316–326
 87. Irfan M N, Oriat C, Groz R. Model inference and testing. In: Advances in Computers. Elsevier, 2013
 88. Kearns M J, Vazirani U V. An Introduction to Computational Learning Theory. MIT Press, 1994
 89. Isberner M, Howar F, Steffen B. The TTT algorithm: a redundancy-free approach to active automata learning. In: Proceedings of International Conference on Runtime Verification. 2014, 307–322
 90. Groce A, Peled D, Yannakakis M. Adaptive model checking. *Logic Journal of the IGPL*, 2006, 14(5): 729–744
 91. Smeenk W, Moerman J, Vaandrager F, Jansen D N. Applying automata learning to embedded control software. In: Proceedings of International Conference on Formal Engineering Methods. 2015, 67–83
 92. Lee D, Yannakakis M. Principles and methods of testing finite state machines-a survey. *Proceedings of IEEE*, 1996, 84(8): 1090–1123
 93. Bos V P, Smeters R, Vaandrager F. Enhancing automata learning by log-based metrics. In: Proceedings of International Conference on Integrated Formal Methods. 2016, 295–310
 94. Chen Y F, Hsieh C, Lengál O, Lii T J, Tsai M H, Wang B Y, Wang F. PAC learning-based verification and model synthesis. In: Proceedings of the 38th International Conference on Software Engineering. 2016, 714–724
 95. Chow T S. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 1978, 3: 178–187
 96. Fujiwara S, Bochmann G V, Khendek F, Amalou M, Ghedamsi A. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 1991, 17(6): 591–603
 97. Shen Y, Lombardi F, Dahbura A T. Protocol conformance testing using multiple UIO sequences. *IEEE Transactions on Communications*, 1992, 40(8): 1282–1287
 98. Vuong S T. The uiof-method for protocol test sequence generation. In: Proceedings of the 2nd IFIP International Workshop on Protocol Test Systems. 1989, 161–175
 99. Aarts F, Jonsson B, Uijen J. Generating models of infinite-state communication protocols using regular inference with abstraction. In: Proceedings of IFIP International Conference on Testing Software and Systems. 2010, 188–204
 100. Aarts F, Jonsson B, Uijen J, Vaandrager F. Generating models of infinite-state communication protocols using regular inference with abstraction. *Journal of Formal Methods in System Design*, 2015, 46(1): 1–41
 101. Cho C Y, Babić D, Shin E C R, Song D. Inference and analysis of formal models of botnet command and control protocols. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. 2010, 426–439
 102. Chalupar G, Peherstorfer S, Poll E, De Ruiter J. Automated reverse engineering using lego®. In: Proceedings of the 8th {USENIX} Workshop on Offensive Technologies ({WOOT} 14). 2014
 103. Vaandrager F. Active learning of extended finite state machines. In: Proceedings of International Conference on Testing Software and Systems. 2012, 5–7
 104. Aarts F, Heidarian F, Kuppens H, Olsen P, Vaandrager F. Automata learning through counterexample guided abstraction refinement. In: Proceedings of International Symposium on Formal Methods. 2012, 10–27
 105. Aarts F, Heidarian F, Vaandrager F. A theory of history dependent abstractions for learning interface automata. In: Proceedings of International Conference on Concurrency Theory. 2012, 240–255
 106. Gold E M. Complexity of automaton identification from given data. *Journal of Information and Control*, 1978, 37(3): 302–320
 107. Isberner M, Steffen B, Howar F. Learnlib tutorial. In: Bartocci E, Majumdar R, eds. Runtime Verification. Springer, Cham, 2015, 358–377
 108. Xiao H. Automatic model learning and its applications in malware detection. Doctoral Dissertation, Nanyang Technological University, 2017
 109. Aarts F D. Tomte: bridging the gap between active learning and real-world systems. Doctoral Dissertation, Radboud University Nijmegen, 2014
 110. Fiterau-Brosteau P. Active model learning for the analysis of network protocols. Doctoral Dissertation, Radboud University, 2018
 111. Irfan M N. Analysis and optimization of software model inference algorithms. Universita de Grenoble, Grenoble, France, 2012
 112. Czerny M X. Learning-based software testing: evaluation of angluin's L* algorithm and adaptations in practice. Batchelors Thesis, Karlsruhe Institute of Technology, 2014
 113. Henrix M. Performance improvement in automata learning. Master thesis, Radboud University, 2015
 114. Uijen J. Learning models of communication protocols using abstraction techniques. Master Thesis, Uppsala University, 2009
 115. Aarts F. Inference and abstraction of communication protocols. Master Thesis, Uppsala University, 2009
 116. Bohlin T, Jonsson B. Regular inference for communication protocol entities. Technical Report 2008-024, Uppsala University, Computer Systems, 2008

117. Berg T. Regular inference for reactive systems. Doctoral Dissertation, Uppsala Universitet, 2006
118. Berg T, Jonsson B, Leucker M, Saksena M. Insights to angluin's learning. *Journal of Electronic Notes in Theoretical Computer Science*, 2005, 118: 3–18
119. Irfan M N. State machine inference in testing context with long counterexamples. In: *Proceedings of International Conference on Software Testing, Verification and Validation*. 2010, 508–511
120. Sidhu D P, Leung T K. Formal methods for protocol testing: a detailed study. *IEEE Transactions on Software Engineering*, 1989, 15(4): 413–426
121. Dorofeeva R, El-Fakih K, Maag S, Cavalli A R, Yevtushenko N. Experimental evaluation of fsm-based testing methods. In: *Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods*. 2005, 23–32
122. Aarts F, Howar F, Kuppens H, Vaandrager F. Algorithms for inferring register automata. In: *Proceedings of International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. 2014, 202–219
123. Stevens P, Moller F. The edinburgh concurrency workbench user manual (version 7.1). Laboratory for Foundations of Computer Science, University of Edinburgh, 1999, 7
124. Maler O, Pnueli A. On the learnability of infinitary regular sets. *Information and Computation*, 1991, 118: 316–326
125. Irfan M N, Oriat C, Groz R. Angluin style finite state machine inference with non-optimal counterexamples. In: *Proceedings of the 1st International Workshop on Model Inference in Testing*. 2010, 11–19
126. Merten M, Howar F, Steffen B, Margaria T. Automata learning with on-the-fly direct hypothesis construction. In: *Proceedings of International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. 2011, 248–260
127. Isberner M, Steffen B. An abstract framework for counterexample analysis in active automata learning. In: *Proceedings of International Conference on Grammatical Inference*. 2014, 79–93
128. Li K, Groz R, Shahbaz M. Integration testing of distributed components based on learning parameterized I/O models. In: *Proceedings of International Conference on Formal Techniques for Networked and Distributed Systems*. 2006, 436–450
129. Howar F. Active Learning of Interface Programs. Doctoral Dissertation, TU Dortmund University, 2012
130. Cassel S, Howar F, Jonsson B, Steffen B. Learning extended finite state machines. In: *Proceedings of International Conference on Software Engineering and Formal Methods*. 2014, 250–264
131. D'Antoni L, Veanes M. The power of symbolic automata and transducers. In: *Proceedings of International Conference on Computer Aided Verification*. 2017, 47–67
132. Isberner M, Howar F, Steffen B. The open-source learnlib. In: *Proceedings of International Conference on Computer Aided Verification*. 2015, 487–495
133. Isberner M, Howar F, Steffen B. The open-source learnlib. In: *Proceedings of International Conference on Computer Aided Verification*. 2015, 487–495
134. Raffelt H, Steffen B. LearnLib a library for automata learning and experimentation. In: *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering*. 2006, 377–380
135. Isberner M, Howar F, Steffen B. Inferring automata with state-local alphabet abstractions. In: *Proceedings of NASA Formal Methods Symposium*. 2013, 124–138
136. Hopcroft J E, Karp R M. A linear algorithm for testing equivalence of finite automata. Technical Report, Cornell University, 1971
137. Giannakopoulou D, Rakamarić Z, Raman V. Symbolic learning of component interfaces. In: *Proceedings of International Static Analysis Symposium*. 2012, 248–264
138. Howar F, Giannakopoulou D, Rakamarić Z. Hybrid learning: interface generation through static, dynamic, and symbolic analysis. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 2013, 268–279
139. Baignczyk A, Schieweck A, Isberner M, Margaria T, Neubauer J, Steffen B. Alex: mixed-mode learning of web applications at ease. In: *Proceedings of International Symposium on Leveraging Applications of Formal Methods*. 2016, 655–671
140. Botinčan M, Babić D. Sigma* symbolic learning of input-output specifications. *Journal of ACM SIGPLAN Notices*, 2013, 48(1): 443–456
141. Bollig B, Katoen J P, Kern C, Leucker M, Neider D, Piegdon D R. Libalf: the automata learning framework. In: *Proceedings of International Conference on Computer Aided Verification*. 2010, 360–364
142. Merten M, Steffen B, Howar F, Margaria T. Next generation learnlib. In: *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2011, 220–223
143. Cassel S, Howar F, Jonsson B. RALib: A learnLib extension for inferring EFSMs. DIFTS, 2015
144. Khalili A, Tacchella A. Aide: automata-identification engine. see Archive. codeplex Website, 2014
145. Aarts F, Schmaltz J, Vaandrager F. Inference and abstraction of the bi-metric passport. In: *Proceedings of International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. 2010, 673–686
146. Aarts F, De Ruiter J, Poll E. Formal models of bank cards for free. In: *Proceedings of Software Testing, Verification and Validation Workshops*. 2013, 461–468
147. Smeenk W. Applying automata learning to complex industrial software. Master's Thesis, Radboud University Nijmegen, 2012
148. Janssen R, Vaandrager F W, Verwer S. Learning a state diagram of tcp using abstraction. Bachelor Thesis, ICIS, Radboud University Nijmegen, 2013, 12
149. Fiterău-Broștean P, Janssen R, Vaandrager F. Learning fragments of the tcp network protocol. In: *Proceedings of International Workshop on Formal Methods for Industrial Critical Systems*. 2014, 78–93
150. Tijssen M. Automatic modeling of ssh implementations with state machine learning algorithms. Bachelor thesis, Radboud University, Nijmegen, 2014
151. Fiterău T P, Janssen R, Vaandrager F. Combining model learning and model checking to analyze tcp implementations. In: *Proceedings of International Conference on Computer Aided Verification*. 2016, 454–471
152. Fiterău-Broștean P, Lenaerts T, Poll E, Ruiter d J, Vaandrager F, Verleg P. Model learning and model checking of SSH implementations. In: *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. 2017, 142–151
153. Schuts M, Hooman J, Vaandrager F. Refactoring of legacy software using model learning and equivalence checking: an industrial experience report. In: *Proceedings of International Conference on Integrated Formal Methods*. 2016, 311–325
154. Neubauer J, Steffen B, Bauer O, Windmüller S, Merten M, Margaria T, Howar F. Automated continuous quality assurance. In: *Proceedings of International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches*. 2012, 37–43
155. Windmüller S, Neubauer J, Steffen B, Howar F, Bauer O. Active continuous quality control. In: *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*. 2013, 111–120
156. Verleg P, Poll E, Vaandrager F. Inferring SSH state machines using protocol state fuzzing. Masters Thesis, Radboud University, 2016
157. Groce A, Peled D, Yannakakis M. Adaptive model checking. In: *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2002, 269–301
158. Li K, Groz R, Shahbaz M. Integration testing of components guided by

- incremental state machine learning. In: *Proceedings of the Testing: Academic and Industrial Conference on Practice and Research Techniques*. 2006, 59–70
159. Shahbaz M, Groz R. Analysis and testing of black-box component-based systems by inferring partial models. *Journal of Software Testing, Verification and Reliability*, 2014, 24(4): 253–288
 160. Shahbaz M, Li K, Groz R. Learning and integration of parameterized components through testing. In: Petrenko A, Veanes M, Tretmans J, Grieskamp W, eds. *Testing of Software and Communicating Systems*. Springer, Berlin, Heidelberg, 2007
 161. Shahbaz M, Parreaux B, Klay F. Model inference approach for detecting feature interactions in integrated systems. In: *Proceedings of International Conference on Feature Interactions in Software and Communication Systems*. 2007, 161–171
 162. Walkinshaw N, Bogdanov K, Derrick J, Paris J. Increasing functional coverage by inductive testing: a case study. In: *Proceedings of International Conference on Testing Software and Systems*. 2010, 126–141
 163. Cobleigh J M, Giannakopoulou D, Pasareanu C S. Learning assumptions for compositional verification. In: *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2003, 331–346
 164. Păsăreanu S C, Giannakopoulou D, Bobaru M G, Cobleigh J M, Barringer H. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 2008, 32(3): 175–205
 165. He F, Mao S, Wang B Y. Learning-based assume-guarantee regression verification. In: *Proceedings of International Conference on Computer Aided Verification*. 2016, 310–328
 166. Chen Y F, Clarke E M, Farzan A, He F, Tsai M H, Tsay Y K, Wang B Y, Zhu L. Comparing learning algorithms in automated assume-guarantee reasoning. In: *Proceedings of International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. 2010, 643–657
 167. Chen Y F, Clarke E M, Farzan A, Tsai M H, Tsay Y K, Wang B Y. Automated assume-guarantee reasoning through implicit learning. In: *Proceedings of International Conference on Computer Aided Verification*. 2010, 511–526
 168. He F, Wang B Y, Yin L, Zhu L. Symbolic assume-guarantee reasoning through BDD learning. In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, 1071–1082
 169. Lin S W, Hsiung P A. Compositional synthesis of concurrent systems through causal model checking and learning. In: *Proceedings of International Symposium on Formal Methods*. 2014, 416–431
 170. Neider D, Topcu U. An automaton learning approach to solving safety games over infinite graphs. In: *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2016, 204–221
 171. Margaria T, Niese O, Steffen B, Erochok A. System level testing of virtual switch (re-) configuration over IP. In: *Proceedings of the 7th IEEE European Test Workshop*. 2002, 67–72
 172. Niese O, Steffen B, Margaria T, Hagerer A, Brune G, Ide H D. Library-based design and consistency checking of system-level industrial test cases. In: *Proceedings of International Conference on Fundamental Approaches to Software Engineering*. 2001, 233–248
 173. Feng L, Lundmark S, Meinke K, Niu F, Sindhu M A, Wong P Y. Case studies in learning-based testing. In: *Proceedings of International Conference on Testing Software and Systems*. 2013, 164–179
 174. Oliveira A L, Silva J P. Efficient algorithms for the inference of minimum size dfas. *Journal of Machine Learning*, 2001, 44(1–2): 93–119
 175. Choi W, Necula G, Sen K. Guided gui testing of android apps with minimal restart and approximate learning. *ACM SIGPLAN Notices*, 2013, 48(10): 623–640
 176. Alur R, Černý P, Madhusudan P, Nam W. Synthesis of interface specifications for java classes. *ACM SIGPLAN Notices*, 2005, 40(1): 98–109
 177. Xiao H, Sun J, Liu Y, Lin S W, Sun C. Tzuyu: learning stateful type-states. In: *Proceedings of the 28th International Automated Software Engineering*. 2013, 432–442
 178. Raffelt H, Steffen B, Margaria T. Dynamic testing via automata learning. In: *Proceedings of Haifa Verification Conference*. 2007, 136–152
 179. Clarke E, Long D, McMillan K. Compositional model checking. In: *Proceedings of the 4th Annual Symposium on Logic in Computer Science*. 1989
 180. Smeenk W, Vaandrager F W. Applying automata learning to complex industrial software. Master's Thesis, Radboud University Nijmegen, 2012
 181. Tappler M, Aichernig B K, Bloem R. Model-based testing IoT communication via active automata learning. In: *Proceedings of IEEE International Conference on Software Testing, Verification and Validation*. 2017, 276–287
 182. Aarts F, Fiterau-Brosteau P, Kuppens H, Vaandrager F. Learning register automata with fresh value generation. In: *Proceedings of International Colloquium on Theoretical Aspects of Computing*. 2015, 165–183
 183. Meinke K, Sindhu M A. Incremental learning-based testing for reactive systems. In: *Proceedings of International Conference on Tests and Proofs*. 2011, 134–151
 184. Volpato M, Tretmans J. Active learning of nondeterministic systems from an ioco perspective. In: *Proceedings of International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. 2014, 220–235
 185. Groz R, Irfan M N, Oriat C. Algorithmic improvements on regular inference of software models and perspectives for security testing. In: *Proceedings of International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*. 2012, 444–457
 186. Ernst M D, Perkins J H, Guo P J, McCamant S, Pacheco C, Tschantz M S, Xiao C. The daikon system for dynamic detection of likely invariants. *Journal of Science of Computer Programming*, 2007, 69(1–3): 35–45
 187. Nachmanson L, Veanes M, Schulte W, Tillmann N, Grieskamp W. Optimal strategies for testing nondeterministic systems. *ACM SIGSOFT Software Engineering Notes*, 2004, 29(4): 55–64
 188. Volpato M, Tretmans J. Approximate active learning of nondeterministic input output transition systems. *Electronic Communications of the EASST*, 2015, 72
 189. Khalili A, Tacchella A. Learning nondeterministic mealy machines. In: *Proceedings of International Conference on Grammatical Inference*. 2014, 109–123
 190. El-Fakih K, Groz R, Irfan M N, Shahbaz M. Learning finite state models of observable nondeterministic systems in a testing context. In: *Proceedings of International Conference on Testing Software and Systems*. 2010, 97–102
 191. Dallmeier V. Mining and checking object behavior. Doctoral Dissertation, 2010
 192. Van der Aalst W M P, Rubin V, Verbeek H M W, Dongen V B F, Kindler E, Günther C W. Process mining: a two-step approach to balance between underfitting and overfitting. *Journal of Software & Systems Modeling*, 2010, 9(1): 87
 193. Meinke K. CGE: a sequential learning algorithm for mealy automata. In: *Proceedings of International Colloquium on Grammatical Inference*. 2010, 148–162
 194. Peled D, Vardi M Y, Yannakakis M. Black box checking. In: Wu J, Chanson S T, Gao Q, eds. *Formal Methods for Protocol Engineering and Distributed Systems*. Springer, Boston, MA, 1999
 195. Cassel S, Howar F, Jonsson B, Merten M, Steffen B. A succinct canonical register automaton model. In: *Proceedings of International Symposium on Automated Technology for Verification and Analysis*. 2011, 366–380



Shahbaz Ali received the MSc degree in Physics from Govt College University, Pakistan in 2000, and the MS degree in computer system engineering from Ghulam Ishaq Khan Institute, Pakistan in 2003. He is currently pursuing the PhD degree in computer software and theory at the School of Computer Science and Engineering, Beihang University, China. His research interest includes pro-

gram analysis, model learning, model-based testing (MBT), learning-based software testing, formal verification, formal methods for software developments.



Hailong Sun received the BS degree in computer science from Beijing Jiaotong University, China in 2001. He received the PhD degree in computer software and theory from Beihang University, China in 2008. He is a professor in the School of Software, Beihang University, China. His research interests include intelligent software engineering, crowd intelligence/crowdsourcing and distributed systems. He is a member of the ACM and the IEEE.



Yongwang Zhao received the PhD degree in computer science from Beihang University, China in 2009. He is an associate professor at the School of Computer Science and Engineering, Beihang University, China. He has also been a Research Fellow in the School of Computer Science and Engineering, Nanyang Technological University, Singapore from 2015. His research interests include

formal methods, OS kernels, information-flow security, and AADL.