Innovations in Systems and Software Engineering (2022) 18:417–426
https://doi.org/10.1007/s11334-022-00449-3

S.I. : ATVA 2021

AALpy: an active automata learning library

Edi Muškardin[1,2] · Bernhard K. Aichernig[1] · Ingo Pill[2] · Andrea Pferscher[1] · Martin Tappler[1,2]

Received: 21 October 2021 / Accepted: 19 February 2022 / Published online: 26 March 2022

Abstract
AALpy is an extensible open-source Python library providing efficient implementations of active automata learning algorithms for deterministic, non-deterministic, and stochastic systems. We put a special focus on the conformance testing aspect in active automata learning, as well as on an intuitive and seamlessly integrated interface for learning automata characterizing real-world reactive systems. In this article, we present AALpy's core functionalities, illustrate its usage via examples, and evaluate its learning performance. Finally, we present selected case studies on learning models of various types of systems with AALpy.

Keywords Active automata learning · Model inference · Testing · Python

1 Introduction

Whenever facing an unknown system, we strive to learn more about its behavior, which in computer science terms often translates to learning its *language*. Regular language inference, a.k.a. automata learning or model mining, is thus a well-studied topic and has been an active field ever since Anguin's seminal paper [6]. Under appropriate abstraction, the input–output traces of a reactive system form a regular language. Consequently, a reactive system can be abstractly modeled as a finite-state machine [17]. For this reason, the topic has gained special interest in the context of model checking [26] and software testing [3] of black-box systems. By providing formal models of black-box systems, automata learning extends the applicability of model-based verification techniques to a class of systems that would otherwise be inaccessible.

✉ Martin Tappler
martin.tappler@ist.tugraz.at

Edi Muškardin
edi.muskardin@student.tugraz.at

Bernhard K. Aichernig
aichernig@ist.tugraz.at

Ingo Pill
ingo.pill@silicon-austria.com

Andrea Pferscher
andrea.pferscher@ist.tugraz.at

[1] Institute of Software Technology, Graz University of Technology, Graz, Austria

[2] TU Graz - SAL DES Lab, Silicon Austria Labs, Graz, Austria

Despite the growing interest, there are few available libraries or frameworks for automata learning. The most notable one is LearnLib [18], an open-source Java library that is the de facto standard when it comes to tools. Compared to LearnLib, our AALpy[1] extends the scope to learning deterministic Moore machines (ONFSMs) and stochastic models. In addition to the support for a wide range of systems, AALpy aims to provide an easy-to-use API.

Due to Python's popularity in software engineering and AI, we chose to implement AALpy in Python such as to target a wide audience, supported also by an open-source MIT license. Especially important for learning models of black-box systems is the fact that Python increasingly serves as interface language for a wide range of software and embedded systems. Popular and influential software, like the machine-learning libraries Keras [9] and PyTorch [25], mainly provide Python APIs, and the Python ecosystem provides a vast amount of libraries, such as Scapy [31] to communicate with and test (embedded) software systems. At the time of writing, Python has just become the most popular programming language according to the TIOBE index October 2021 [40].

This article is an extended version of our tool paper [22] presented at the 19^{th} International Symposium on Automated Technology for Verification and Analysis (ATVA 2021). Additional content presented in this article covers applications of AALpy in several case studies (Sect. 4), an experimental comparison with LearnLib [18], and extended

[1] Code, documentation, interactive examples, and a comprehensive Wiki can be found at https://github.com/DES-Lab/AALpy.

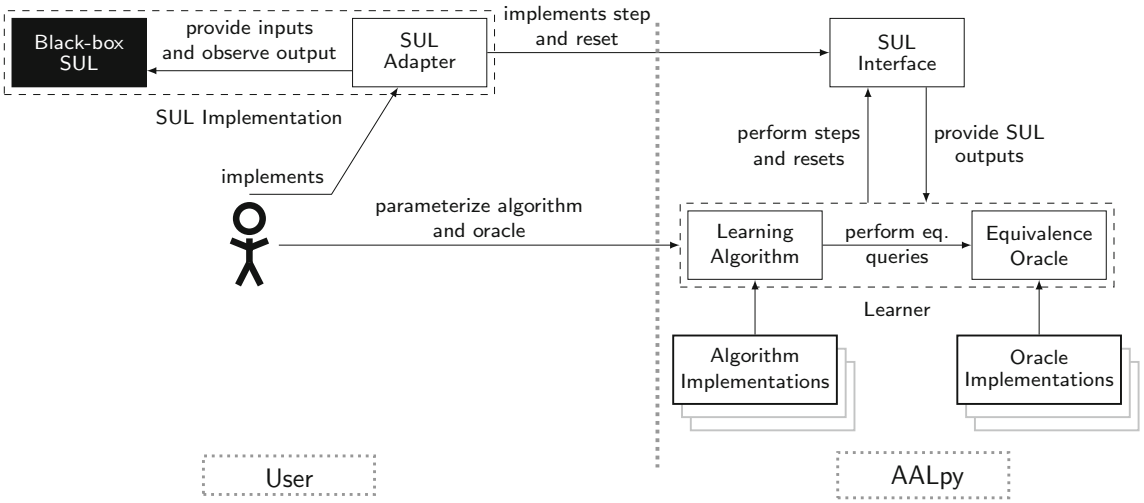


Fig. 1 AALpy's Interface and structure

Listing 1 Learning regural expressions with AALpy

```
class RegexSUL(SUL):
  # System under learning for regular expressions
  def __init__(self, regex: str):
    super().__init__()
    self.regex = regex if regex[-1] == '$' else regex + '$'
    self.test_string = ""

  def pre(self):
    self.test_string = ""

  def post(self):
    pass

  def step(self, letter):
    if letter is not None:
      self.test_string += str(letter)
    return True if re.match(self.regex, self.test_string) else False

regex = '((0|1)*0)*1(11)*(0(0|1)*1)*0(00)*(1(0|1)*)*' \\# complement of Tomita 3 grammar
alphabet = [0, 1]
regex_sul = RegexSUL(regex)
eq_oracle = RandomWMethodEqOracle(alphabet, regex_sul)
learned_automaton = run_Lstar(alphabet, regex_sul, eq_oracle,  automaton_type='dfa')
visualize_automaton(learned_automaton)
```

descriptions of AALpy's features, such as the supported learning algorithms and equivalence oracles, both of which have been extended since the presentation at ATVA 2021.

2 AALpy – Intuitive automata learning in python

Key features of our library are its modular design, a seamlessly integrated deployment process, and support for learning various types of system models. Efficient implementations of state-of-the-art learning algorithms for deterministic, non-deterministic, and stochastic automata paired with efficient conformance testing enables automata learning in a wide variety of environments. AALpy's accessibility and usability are enhanced via extensive documentation and multiple demonstrating examples for each of the library's functionalities—complemented by visualization and logging capabilities. The latter may be of special interest for educational purposes.

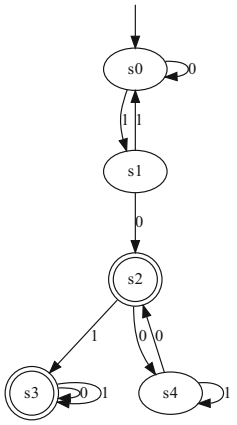
The query-based automata learning algorithms implemented in AALpy are based on the minimally adequate teacher (MAT) framework by Angluin [6]. We particularly focus on learning models of reactive systems, whose

input–output behavior under appropriate abstraction can be captured by regular languages. Learning models of such systems in the MAT framework lends itself nicely to a test-based implementation, as demonstrated in various case studies [5,11,30,35]. Algorithms in this framework alternate between two phases. In an exploitation phase, membership queries are issued to gain new information about the SUL related to known data. At the end of such a phase, a hypothesis automaton is formed from the queried data. The hypothesis and the SUL are checked against each other in an exploration phase via so-called equivalence queries. These queries shall return counterexamples to equivalence between the SUL and the current hypothesis to falsify the latter. A counterexample serves to refine the hypothesis and to progress learning. Learning terminates with the final learned hypothesis as output once it is not possible to falsify said hypothesis, that is, an equivalence query returns that SUL and hypothesis are equivalent.

When learning models of reactive systems, membership queries ask for the outputs produced by the SUL in response to a given sequence of inputs. From a testing point of view, such queries can be implemented through a single test of the SUL with the query inputs. There is no test verdict, but the SUL outputs are recorded by the learning algorithm. In test-based automata learning, equivalence queries are implemented via conformance testing—we refer to implementations as equivalence oracles. Conformance testing derives a set of test cases from the current hypothesis and executes them on both SUL and hypothesis. A test case revealing a difference in their input–output behavior is a witness of inequivalence. Like a membership query, a test case essentially asks for the SUL outputs produced in response to a sequence of inputs. Thus, we uniformly refer to both as queries, while using more specific terms, like "equivalence query," where necessary.

The active approach to learning in the MAT framework combines well with online testing, where test-case execution proceeds in a step-wise manner. At the beginning of each query, the SUL is reset to a known initial state. Then, each input is performed as an individual step, with the output produced by the SUL being the result of the step. The final result of a query is the sequence of outputs produced in a sequence of steps.

To this end, AALpy interfaces the SUL and a selected learning algorithm via a step-based interface. Thus, in an individual step, an input stimulus is provided to the SUL and then the resulting output is observed. For real-world SULs, interfacing the SUL and the algorithm may involve some abstraction and concretization, for instance, implemented via a mapper [1]. When employing AALpy, a user thus in principle only has to define the functionality for a step, as well as a proper reset for the SUL in order to be able to start queries from a known initial state. AALpy implements queries as sequences of steps and resets. If required, a user can implement queries directly.




```
Hypothesis 1: 1 states.
Hypothesis 2: 4 states.
Hypothesis 3: 5 states.
--------------------------------
Learning Finished.
Learning Rounds:  3
Number of states: 5
Time (in seconds)
Total                : 0.01
  Learning algorithm   : 0.0
  Conformance checking : 0.01
Learning Algorithm
  # Membership Queries  : 20
  # MQ Saved by Caching : 18
  # Steps               : 77
Equivalence Query
  # Membership Queries  : 100
  # Steps               : 1169
--------------------------------
```

Fig. 2 Output of Listing 1 Showing the visualization of the learned automaton (left) and learning statistics (right).

When employing AALpy, a user follows a three-stage process:

(a) define the SUL interface for the learning engine,
(b) select an equivalence oracle, and
(c) select, customize, and run the learning algorithm.

In (a), three methods are to be defined: `pre`, `post`, and `step` (see also Listing 1). With `pre`, we initialize and setup the SUL, while `post` shall support a graceful shutdown/memory cleanup. As informally suggested above, `step` encapsulates a single step in the query execution, such that formally some $\sigma \in \Sigma$ from the input alphabet Σ is mapped to a concrete input/or action for the SUL, and the SUL's output is observed and reported back as a letter γ in some output alphabet Γ. Note that we do not limit alphabets to integers, characters, or strings. In particular, Σ and Γ can be lists of hashable objects, or even class methods with appropriate arguments.

In (b), the user selects and parameterizes one of the equivalence oracles. The choice of oracle and its parameters will determine the amount and type of testing performed in the equivalence query. Hence, this oracle configuration can be performed based on the available testing budget. More details on available oracles can be found in Sect. 2.4.

Finally, in (c) the user provides parameters to the appropriate learning algorithm. Some of the common parameters are the maximum number of learning rounds, a counterexample processing strategy, and the amount of information printed during the learning process. Other parameters vary based on the chosen algorithm. Available learning algorithms are described in the remainder of the section.

The tree-stage setup process, as well as the overall high-

level library architecture, can be seen in Fig. 1. The user implements an SUL adapter with the three SUL-interface methods described above, that is, `step`, `pre`, and `post`. Additionally, the user configures a learning algorithm and an equivalence oracle that interface with the SUL via the SUL interface.

Example 1. Learning a regular expression. Listing 1 implements active learning of a DFA conforming to a regular expression.

In Lines 1-17, we show a simple SUL that parses any regular expression. In Lines 19 and 20, we define a regular expression over a binary alphabet. In Line 22, we select the equivalence oracle used for answering equivalence queries via conformance testing, and in Line 23 we select the learning algorithm and execute it. When finished, AALpy prints the learning statistics and visualizes the automaton as shown in Fig. 2.

2.1 Learning deterministic models

Let us now describe the supported learning algorithms, starting with the support of **deterministic learning of DFA, Mealy and Moore machines**.

We extended the original L^* algorithm [6] with two counterexample processing techniques [29,32]. Both techniques extract so-called distinguishing suffixes from counterexamples. These are sequences that distinguish two states of the SUL that map to the same state in an intermediate learned hypothesis automaton, thus revealing an error in the hypothesis. The first technique [29] analyzes the counterexample and finds a single distinguishing suffix at the cost of a logarithmic number of queries with respect to counterexample length, while the latter finds the distinguishing suffix that will avoid consistency violations without posing any queries. As reported in our previous work [4], counterexample processing is essential for efficient learning.

In addition, AALpy implements query caching. The cache reduces the number of SUL interactions performed for membership queries. It encodes membership query results as a tree that is updated during learning as well as equivalence checking. Via this cache we can avoid posing duplicate membership queries and membership queries for prefixes of already seen traces.

2.2 Learning non-deterministic models

The assumption of deterministic SUL behavior limits the applicability of active automata learning. Non-determinism might result from input and output alphabet abstraction or from ignoring system properties, such as timed behavior. To manage such circumstances, AALpy also offers learning algorithms for non-deterministic and stochastic systems.

AALpy provides two algorithms for learning **observable non-deterministic finite-state machines (ONFSM)**. These algorithms assume observable non-deterministic SUL behavior, meaning that the SUL may produce outputs non-deterministically, while non-deterministic state changes are only possible with different outputs.

The notion of observable non-determinism should not be confused with our general black-box view. We cannot observe the system state directly, but we assume that there is a uniquely defined target state for each triple of source state, input, and output.

The first learning-algorithm implementation follows the proposed learning algorithm of El-Fakih et al. [10]. However, this algorithm is based on an "all-weather condition," that is, all possible outputs can be observed immediately. AALpy replaces this assumption with a more practical implementation using sampling. Recently, Pferscher and Aichernig [27] proposed an extension of the classic ONFSM learning algorithm. Their extension learns abstracted ONFSMs by introducing equivalence classes for outputs. This abstraction mechanism enables the creation of smaller models and faster learning.

2.3 Learning stochastic models

AALpy's support of **active learning of stochastic systems** draws on L^*_{MDP} [36,38] and L^*_{SMM} [39], an improved adaptation of L^*_{MDP}. The learning algorithms formalize the behavior of stochastic systems as either (SMMs) or (MDPs). Both types of models can be controlled by its environment through inputs and react stochastically through state changes and by producing outputs.

While the previously discussed learning approaches rely on membership and equivalence queries, L^*_{SMM} implements a "stochastic" teacher that is able to answer *tree queries* and *equivalence queries*. Tree queries serve the same purpose as membership queries in gathering additional information on the SUL's behavior. Stochastic behavior makes it inefficient to ask membership queries on individual sequences s, since the SUL may or may not produce s or any of its prefixes. Asking for information related to a tree created by merging a set of sequences accounts for that. Compared to the original implementation of L^*_{MDP} [34], L^*_{SMM} as available in AALpy requires fewer parameters and is more robust to sparse observations. In practice, users only have to implement the `SUL` interface as discussed in Sect. 2. That is, there are no additional requirements on stochastic SULs.

Models learned by L^*_{SMM} converge to the canonical model underlying the input–output behavior of the SUL. To the best of our knowledge, there are currently no automata learning

algorithms for MDPs and similar formalisms[2] that provide accuracy guarantees for models learned from finite samples of system traces. Moreover, different learning algorithms create models with different properties, even though they may converge to the same models in the limit. For instance, we observed in previous work [38] that IOALERGIA [20] creates smaller models than L^*-based learning. Such models may be desirable in certain application scenarios, as well as the fact that IOALERGIA learns passively from given traces. For this reason, AALPY implements ALERGIA [20], adding support for **passive learning of Markov Chains and MDPs**. Given traces for passive learning can be extended through active learning extensions of ALERGIA [2,8]. We are currently working on adding one of the extensions to AALPY, which adds support for **probabilistic black-box reachability checking** [2].

2.4 Conformance testing

We address equivalence queries via conformance testing. As outlined at the beginning of this section, we apply conformance testing to check whether a hypothesis automaton is equivalent to an SUL. To this end, we generate a test suite from the hypothesis and execute it on both the SUL and the current hypothesis. A test case revealing a difference between them serves as a counterexample to equivalence. Most equivalence oracles available in AALPY apply the guiding principle suggested by Howar et al.: Equivalence checking in automata learning should try "finding counterexamples fast" instead of "trying to prove equivalence" between the SUL and a hypothesis [16]. Therefore, we focus on efficient random-testing heuristics rather than expensive deterministic conformance testing, such as the W-method. AALPY provides eleven equivalence oracles, and new ones can be added easily. To this end, AALPY supports a user by providing a (not necessarily minimal) characterization set of the hypothesis, a shortest path to each state, and a set of previously observed traces (cache). Currently, AALPY implements the following equivalence oracles:

- *W-method:* Formal testing method of proving equivalence between an implementation and a specification FSM up to predefined maximum number of implementation states. Here, a hypothesis automaton serves as specification for the purpose of test-case generation.
- *Random word:* Test cases consist of a sequence of random inputs of uniformly distributed length.
- *Random walk:* Test cases consist of a sequence of random inputs with geometric length distribution.
- *Random W-method:* Each test case consists of a prefix to a randomly chosen state, a random walk, and a random element of the characterization set of the current hypothesis.
- *Probably approximately correct (PAC) oracle*: Random-word-based oracle providing the guarantee that the returned hypothesis is an ϵ-approximation of the correct hypothesis with the probability of at least 1 - δ. This is achieved by setting the number of test cases in the learning round r is defined as $\frac{1}{\epsilon} \times (\log(\frac{1}{\delta}) + r \times \log(2))$, where ϵ is the generalization error and δ the confidence [21].
- *Fixed prefix random walk:* Test cases consists of a prefix to a randomly chosen state and a random walk.
- *Cache-tree based exploration:* Each test case corresponds to a path from the root of the cache to one of its leaves concatenated with a random walk. In this way we extend the boundary of the already explored search space.
- *k-Way transition coverage:* Selects test cases based on random testing and optimizing k-way transitions coverage of the hypothesis. The oracle follows a two-step process, in which it first generates a large number of random walks. In the second step, it greedily selects a subset of these tests to optimize coverage.
- *Transition/same state focus:* Each test case is created by a guided random walk. Based on a parameter ϵ, each input either leads to the same state with a probability of ϵ or to a new state with a probability of $1 - \epsilon$.
- *Breath-first exploration:* This oracle creates test cases through a complete breadth-first exploration up to predefined depth.
- *User input oracle:* Interactive oracle in which a user provides inputs and obtains the corresponding outputs from the SUL and the current hypothesis.
- *Eq. Oracles for Stochastic Setting:* AALPY implements random walk and random word equivalence oracles for the stochastic setting. Aside from finding counterexamples, they also update the hypothesis based on observed input–output pairs.

We refer the interested reader to AALPY's documentation and Wiki[3] for more detailed descriptions, suggested use cases, and parameter explanations for each of these oracles.

2.5 Additional features

For an enhanced user experience, AALPY can save learned automata to files following the community's syntax [24],

[2] Note that we generally consider models that are controllable through non-deterministically chosen inputs that support active software testing. Learning of uncontrollable models, like probabilistic finite automata, has been studied in the PAC framework [7].

[3] https://github.com/DES-Lab/AALpy/wiki.

visualize them, and display information about the learning progress and the observation table. AALpy implements several data parsers easing the passive learning process with Alergia. For evaluation, a user may generate random automata, define them as an SUL and then learn them. For verification of stochastic systems, AALpy provides a translation of MDPs into the format of the probabilistic model checker Prism [19].

3 Experimental evaluation

In order to showcase AALpy's performance, we conducted several experiments on a Dell Lattitude 5410 with an Intel Core i7-10610U processor, 8 GB of RAM running Windows 10 and using PyPy[4] 3.9. In particular, we experienced a performance benefit of using PyPy over CPython.[5]

Learning of deterministic models. The efficiency of AALpy for learning deterministic models was evaluated with extensive experiments on random automata. We conducted two types of experiments, one in which we increased the number of states of the target automata while keeping the size of the input alphabet constant, and one where we increased the size of the input alphabet whilst keeping the size of the target automata constant. Each experiment was repeated 20 times to obtain average values. Figure 3 shows the results. We observed that the automaton size affects DFA learning more than Mealy machine learning. On the other hand, DFA learning is least affected by the increase in the input alphabet. Furthermore, we see that the runtime increases linearly with the number of states and almost linearly with the size of the alphabet. We also performed experiments on learning random Moore machine, where we observed similar behavior as for Mealy machines; therefore, we do not include the results in the figures.

To compare with the state of the art in active automata learning, both experiments were repeated with Learnlib [18], with the results of these experiments being shown in Fig. 3. Our findings are consistent with those presented by LearnLib's developers [18]. We observe that learning of random automata is slightly faster with LearnLib. This minor difference can be attributed to the execution speed differences between statically and dynamical typed languages and potentially differences in internal data structures. However, AALpy performed slightly better on DFA with bigger alphabets.

These experiments ignore SUL interaction time, which is the most resource-intensive part of the learning process on non-simulated systems, such as network protocols [11,35]. To account for that, we performed a second experiment where we compared the number of learning steps and the actual learning time needed to learn systems requiring an assumed time of 25 milliseconds to complete a learning step. The results of the experiments with both, AALpy and LearnLib, are shown in Fig. 4. We observe that both libraries required similar numbers of steps to learn the complete model of the system. Under the assumption that each step requires a constant time of 25 milliseconds to execute, the runtime differences of the learning-algorithm implementations shown in Fig. 3 become negligible compared to the system-interaction time. This can be attributed to the usage of equivalent algorithms, with minor differences in the numbers of steps due to randomness found in equivalence oracles. We conclude that there is no practical difference in speed between AALpy and LearnLib for learning in practice.

Learning of stochastic models. We evaluated AALpy on learning stochastic models with the same experiments as the original Java version of L^*_{MDP} [38]. That is, we learned MDPs by simulating known ground-truth MDP models as black boxes and measured the learning runtime and accuracy. To measure accuracy, we used a probabilistic model-checker to compute probabilities for satisfying temporal properties with the ground-truth models and the learned models. The model-checking error then quantifies accuracy, which we compute as the absolute difference between the results on the ground truth and the results on the learned models. Figure 5 shows the average runtime and the average model-checking errors measured in the experiments. We can see that AALpy and the Java implementation are generally similarly fast and produce similarly accurate models. Evaluation differences can be attributed to minor implementation details.

4 Applications of AALpy

Since AALpy's first release in April 2021, we and others have used AALpy in a number of applications spanning various application domains and fields of research related to testing. The variety of domains highlights the flexibility and ease of use of AALpy as well as the potential of rapid development of testing tools in a Python environment. In this section, we provide an overview of these applications.

4.1 Fuzzing Bluetooth low energy

Automata learning proved itself as a useful technique to analyze communication protocols, e.g., MQTT [35], SSH [12], TCP [11], TLS [13,30], or the 802.11 4-Way Handshake [33]. The literature frequently denotes learning-based testing techniques on communication protocols as state fuzzing. Recently, Pferscher and Aichernig [28] used AALpy to learn the connection interface of BLE devices. Using a learning library implemented in Python creates the opportunity

[4] https://www.pypy.org/

[5] https://github.com/python/cpython.

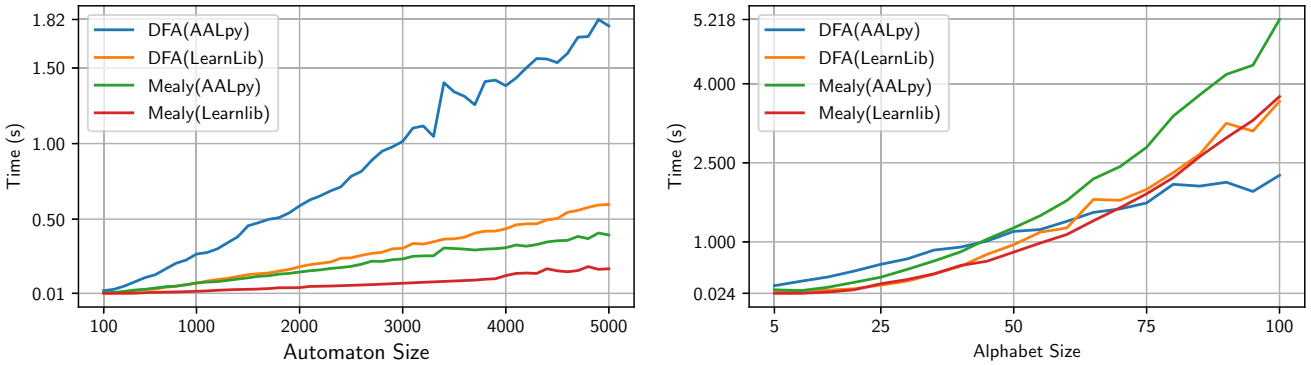
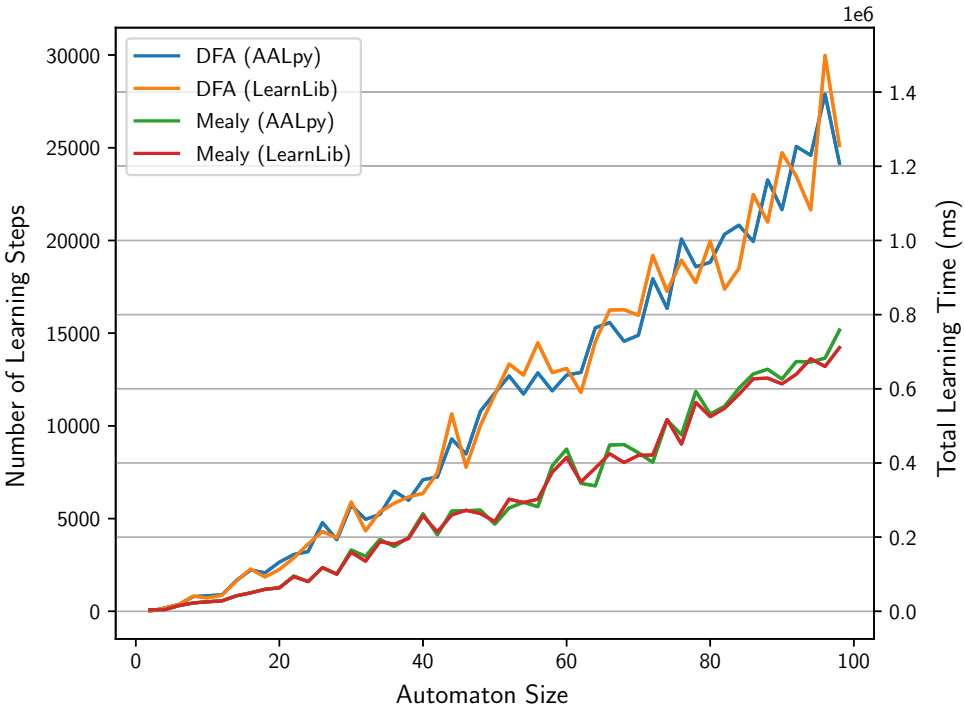


Fig. 3 Runtime of the deterministic L^* with respect to automata size (for an alphabet of size 10) and alphabet size (for an automaton with 1000 states. Interaction time with the SUL is minimal as learning is performed on simulated systems.

Fig. 4 Comparison between AALpy and LearnLib with respect to the number of steps performed and total runtime during automata learning of a deterministic system. Each step on the system takes 25ms to complete.



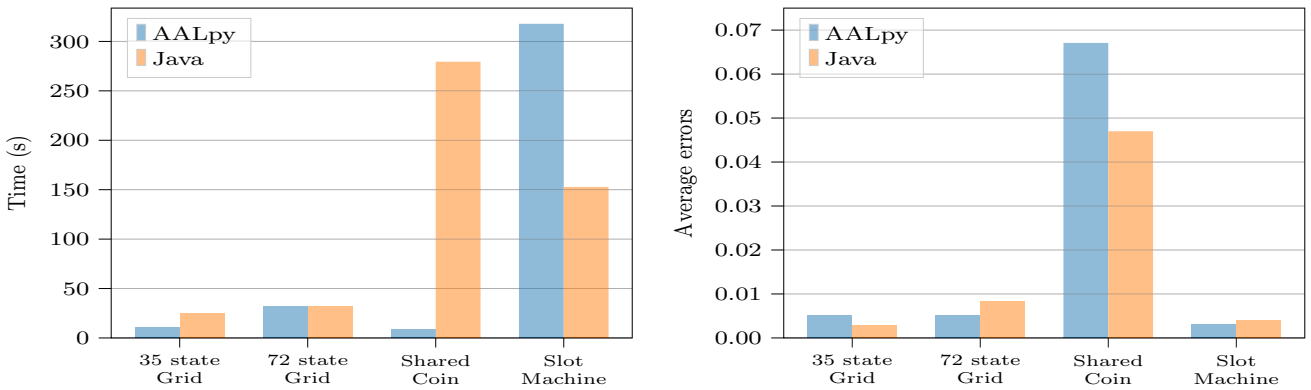


Fig. 5 Runtime measurements and probabilistic model-checking errors on learned models for the AALpy implementation and the Java implementation of L^*_{MDP}.

for a smooth integration of handy communication package libraries like SCAPY [31]. In this application, SCAPY was used to construct BLE packages also on lower levels of the BLE protocol stack. Furthermore, the case study on the BLE protocol shows that AALPY can be extended by a fault-tolerant interface to the SUL. Considering fault tolerance is especially necessary in the learning of communication protocols since requests or responses might be delayed or lost. Additionally, AALPYs caching mechanism reduces the costs of time-expensive network communication. In their presented case study, they learned the behavioral models of five BLE devices. They also discussed countermeasures in the case of non-deterministic behavior. The learned models were different for every device. Considering the differences in the behavioral models, a fingerprinting sequence could be generated that uniquely identifies the BLE device. In future work, the learned models can be used to generate a stateful black-box fuzzing technique as proposed by Aichernig et al. [5].

4.2 Model-based diagnosis

Model-based diagnosis is a technique that detects and isolates the causes of faults. However, the lack of a diagnostic model often prohibits us from deploying diagnostic reasoning for reasoning about the root causes of encountered issues. In [23], we examined how to exploit active automata learning for learning deterministic and stochastic models from black-box reactive systems for diagnostic purposes.

With AALPY, we can learn models of faulty systems for being able to deploy model-based reasoning. Furthermore, we showed how to exploit fault models in the learning process, such as to derive a behavioral model describing the entire corresponding diagnosis search space.

4.3 Extracting models from recurrent neural networks

We applied AALPY to extract automata out of recurrent neural networks that have been trained to recognize regular languages.[6] In particular, we observed that sufficient allocation of testing resources in the equivalence check will lead to counterexamples that state-of-the-art white-box methods were unable to find. This further reinforces the need for the development of advanced equivalence checking testing techniques.

Furthermore, we showed how learning-based testing can be used to extended the RNNs training set by obtaining new samples from the ground truth model and how a mapper can be used to learn abstracted models of RNN's input–output behavior.

4.4 Finding bugs in VIM

AALPY has been used as a debugger tool for software that is internally based on a state machine, more specifically for the text editor Vim and its feature-enriched fork Neovim. A group of researchers used AALPY to generate a graph of newly introduced modes, and during the learning process encountered non-determinism. After the examination of non-deterministic sequences, they were able to isolate the root causes and submit a bug report. The bugs found were later fixed by the community[7].

5 Conclusion

We presented AALPY, the first active automata learning library implemented in Python. AALPY efficiently learns deterministic, non-deterministic, and stochastic systems. AALPY provides its users with a set of equivalence oracles, different configurations of learning algorithms, and the ability to visualize the learning process and results. AALPY has been successfully used to learn the protocols of MQTT and Bluetooth. These learned models serve as a basis for learning-based testing [3] and fuzzing [5].

AALPY is for researchers, educators, and industry alike. Its modular design provides a solid basis for experimentation with new learning algorithms, equivalence oracles, and counterexample processing. In future, we intend to extend these functionalities, with SAT-based learning [15] and learning without reset [14]. We hope that the community will recognize AALPY as an attractive foundation for further research, and welcome suggestions and extensions.

Funding Open access funding provided by Graz University of Technology.

[6] https://github.com/DES-Lab/Extracting-FSM-From-RNNs.

[7] https://github.com/DES-Lab/AALpy/discussions/13.

References

1. Aarts F, Jonsson B, Uijen J (2010) Generating models of infinite-state communication protocols using regular inference with abstraction. In: ICTSS 2010
2. Aichernig BK, Tappler M (2019) Probabilistic black-box reachability checking (extended version). Formal Methods Syst Des 54(3):416–448
3. Aichernig BK, Mostowski W, Mousavi MR, Tappler M, Taromirad M (2018) Model learning and model-based testing. In: Bennaceur A, Hahnle R, Meinke K (eds) Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers, Springer, Lecture Notes in Computer Science, vol 11026, pp 74–100. https://doi.org/10.1007/978-3-319-96562-8_3
4. Aichernig BK, Tappler M, Wallner F (2020) Benchmarking combinations of learning and testing algorithms for active automata learning. TAP 2020:3–22
5. Aichernig BK, Muškardin E, Pferscher A (2021) Learning-based fuzzing of IoT message brokers. In: 14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021, IEEE, pp 47–58. https://doi.org/10.1109/ICST49551.2021.00017
6. Angluin D (1987) Learning regular sets from queries and counterexamples. Inf Comput 75(2):87–106
7. Castro J, Gavalda R (2016) Learning probability distributions generated by finite-state machines. In: Heinz J, Sempere JM (eds) Topics in Grammatical Inference, Springer, Berlin, Heidelberg, pp 113–142. https://doi.org/10.1007/978-3-662-48395-4_5
8. Chen Y, Nielsen TD (2012) Active learning of Markov decision processes for system verification. In: 11th International Conference on Machine Learning and Applications, ICMLA, Boca Raton, FL, USA, December 12-15, 2012. Volume 2, IEEE, pp 289–294. https://doi.org/10.1109/ICMLA.2012.158
9. Chollet F, et al. (2015) Keras. https://github.com/fchollet/keras
10. El-Fakih K, Groz R, Irfan MN, Shahbaz M (2010) Learning finite state models of observable nondeterministic systems in a testing context. ICTSS 2010:97–102
11. Fiterau-Brostean P, Janssen R, Vaandrager FW (2016) Combining model learning and model checking to analyze TCP implementations. In: Chaudhuri S, Farzan A (eds) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II, Springer, Lecture Notes in Computer Science, vol 9780, pp 454–471. https://doi.org/10.1007/978-3-319-41540-6_25
12. Fiterau-Brostean P, Lenaerts T, Poll E, de Ruiter J, Vaandrager FW, Verleg P (2017) Model learning and model checking of SSH implementations. In: Erdogmus H, Havelund K (eds) Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017, ACM, pp 142–151. https://doi.org/10.1145/3092282.3092289
13. Fiterau-Brostean P, Jonsson B, Merget R, de Ruiter J, Sagonas K, Somorovsky J (2020) Analysis of DTLS implementations using protocol state fuzzing. In: Capkun S, Roesner F (eds) 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020, USENIX Association, pp 2523–2540. https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean
14. Groz R, Bremond N, Simao A, Oriat C (2020) hW-inference: A heuristic approach to retrieve models through black box testing. JSS 159:110426
15. Heule MJH, Verwer S (2010) Exact DFA identification using SAT solvers. In: Sempere JM, Garcia P (eds) ICGI 2010, pp 66–79
16. Howar F, Steffen B, Merten M (2010) From ZULU to RERS – lessons learned in the ZULU challenge. In: ISoLA 2010, LNCS, vol 6415, pp 687–704
17. Hungar H, Niese O, Steffen B (2003) Domain-specific optimization in automata learning. In: Jr WAH, Somenzi F (eds) Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings, Springer, Lecture Notes in Computer Science, vol 2725, pp 315–327. https://doi.org/10.1007/978-3-540-45069-6_31
18. Isberner M, Howar F, Steffen B (2015) The open-source LearnLib – a framework for active automata learning. In: CAV 2015 (I), LNCS, vol 9206, pp 487–495
19. Kwiatkowska MZ, Norman G, Parker D (2011) PRISM 4.0: Verification of probabilistic real-time systems. In: CAV 2011, LNCS, vol 6806, pp 585–591
20. Mao H, Chen Y, Jaeger M, Nielsen TD, Larsen KG, Nielsen B (2016) Learning deterministic probabilistic automata from a model checking perspective. Mach Learn 105(2):255–299
21. Mohri M, Rostamizadeh A, Talwalkar A (2012) Foundations of Machine Learning. Adaptive computation and machine learning, MIT Press. http://mitpress.mit.edu/books/foundations-machine-learning-0
22. Muskardin E, Aichernig BK, Pill I, Pferscher A, Tappler M (2021) Aalpy: An active automata learning library. In: Hou Z, Ganesh V (eds) Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceedings, Springer, Lecture Notes in Computer Science, vol 12971, pp 67–73. https://doi.org/10.1007/978-3-030-88885-5_5
23. Muškardin E, Pill I, Tappler M, Aichernig BK (2021) Automata learning enabling model-based diagnosis. In: 32nd International Workshop on Principle of Diagnosis, Hamburg-Germany, September 13th-15th
24. Neider D, Smetsers R, Vaandrager F, Kuppens H (2019) Benchmarks for automata learning and conformance testing. In: Models, Mindsets, Meta: The What, the How, and the Why Not?, LNCS, vol 11200, pp 390–416
25. Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, Desmaison A, Kopf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai J, Chintala S (2019) Pytorch: An imperative style, high-performance deep learning library. In: Wallach HM, Larochelle H, Beygelzimer A, d'Alche-Buc F, Fox EB, Garnett R (eds) Advances in Neural Information Processing Systems 32, Curran Associates, Inc., pp 8024–8035
26. Peled DA, Vardi MY, Yannakakis M (2002) Black box checking. J Autom Lang Comb 7(2):225–246
27. Pferscher A, Aichernig BK (2020) Learning abstracted non-deterministic finite state machines. In: Casola V, Benedictis AD, Rak M (eds) Testing Software and Systems - 32nd IFIP WG 6.1 International Conference, ICTSS 2020, Naples, Italy, December 9-11, 2020, Proceedings, Springer, Lecture Notes in Computer Science, vol 12543, pp 52–69. https://doi.org/10.1007/978-3-030-64881-7_4
28. Pferscher A, Aichernig BK (2021) Fingerprinting Bluetooth Low Energy devices via active automata learning. In: Formal Methods - 24th International Symposium, FM 2021, Beijing, China, November 20-26, 2021, Accepted, Springer
29. Rivest RL, Schapire RE (1993) Inference of finite automata using homing sequences. Inform. Comput. 103(2):299–347
30. de Ruiter J, Poll E (2015) Protocol state fuzzing of TLS implementations. In: Jung J, Holz T (eds) 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015, USENIX Association, pp 193–206. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter
31. Scapy (2021) Scapy. https://github.com/secdev/scapy/, Accessed Sept 10 2021

32. Shahbaz M, Groz R (2009) Inferring Mealy machines. In: FM 2009, LNCS, vol 5850, pp 207–222
33. Stone CM, Chothia T, de Ruiter J (2018) Extending automated protocol state learning for the 802.11 4-way handshake. In: opez J, Zhou J, Soriano M (eds) Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I, Springer, Lecture Notes in Computer Science, vol 11098, pp 325–345. https://doi.org/10.1007/978-3-319-99073-6_16
34. Tappler M (2019) Evaluation material for L^*-based learning of Markov decision processes (37). Available via https://doi.org/10.6084/m9.figshare.7960928.v1, Accessed Sept 10 2021
35. Tappler M, Aichernig BK, Bloem R (2017) Model-based testing IoT communication via active automata learning. In: 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017, IEEE Computer Society, pp 276–287. https://doi.org/10.1109/ICST.2017.32
36. Tappler M, Aichernig BK, Bacci G, Eichlseder M, Larsen KG (2019a) L*-based learning of Markov decision processes. In: ter Beek MH, McIver A, Oliveira JN (eds) Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings, Springer, Lecture Notes in Computer Science, vol 11800, pp 651–669. https://doi.org/10.1007/978-3-030-30942-8_38
37. Tappler M, Aichernig BK, Bacci G, Eichlseder M, Larsen KG (2019b) L^*-based learning of Markov decision processes. In: FM 2019, LNCS, vol 11800, pp 651–669
38. Tappler M, Aichernig BK, Bacci G, Eichlseder M, Larsen KG (2021a) L*-based learning of Markov decision processes (extended version). FAOC
39. Tappler M, Muškardin E, Aichernig BK, Pill I (2021b) Active learning of stochastic reactive systems. In: Software Engineering and Formal Methods - 19th International Conference, SEFM 2021, Lecture Notes in Computer Science
40. Tiobe (2018) https://www.tiobe.com/tiobe-index/, Accessed Sept 10 2021