`libalf`: the Automata Learning Framework*

Benedikt Bollig[1], Joost-Pieter Katoen[2], Carsten Kern[2], Martin Leucker[3], Daniel Neider[2], and David R. Piegdon[2]

[1] LSV, ENS Cachan, CNRS, [2] RWTH Aachen University, [3] TU München

Abstract. This paper presents `libalf`, a comprehensive, open-source library for learning formal languages. `libalf` covers various well-known learning techniques for finite automata (e.g. Angluin's L*, Biermann, RPNI etc.) as well as novel learning algorithms (such as for NFA and visibly one-counter automata). `libalf` is flexible and allows facilely interchanging learning algorithms and combining domain-specific features in a plug-and-play fashion. Its modular design and C++ implementation make it a suitable platform for adding and engineering further learning algorithms for new target models (e.g., Büchi automata).

1 Introduction

The common objective of all learning algorithms is to generalize knowledge gained throughout a learning process. In such a process, the learning algorithm is confronted with classified examples. They are utilized to derive some kind of hypothesis which is able to classify new examples in conformance with the examples already seen. Typically, learning algorithms are grouped into *online* and *offline* algorithms. Online learning techniques are capable of actively asking queries to some kind of *teacher* who is able to classify these queries. Offline algorithms, on the other hand, are passively provided with a set of classified examples from which they have to build an apposite hypothesis.

In recent years, learning algorithms have become increasingly popular for various application domains and have been successfully used in different fields of computer science, reaching from robotics over pattern recognition (e.g., in bioinformatics) to natural language recognition. Especially in the area of automatic verification, learning techniques have proved their great usefulness. They were used for minimizing partially specified systems [1], model checking blackbox systems (e.g., [2]), and for improving regular model checking (e.g., [3]). To put it bluntly, automata learning is en vogue.

The need for a unifying framework collecting various types of learning techniques is, thus, beyond all questions. In addition, it is desirable to have possibilities of easily exchanging or extending the implemented learning algorithms to compare assets and drawbacks for certain user applications. For users' convenience a library should provide additional features, such as means for statistical evaluation or loggers. Unfortunately, existing learning frameworks only partly cover these requirements.

The main objective of this paper is to present a new library called the *automata learning framework* (`libalf` for short). `libalf` unifies different kinds of learning techniques into a single flexible and easy-to-extend library with a clearly structured user interface. We would like `libalf` to become a comprehensive compendium of learning techniques to which everybody has access and can contribute in a public domain fashion.

* This work is partially supported by the DAAD (Procope 2009).

2 Related work

A large number of learning algorithms can be found in the literature. Usually, the most important and influential ones are implemented again and again, but often as *quick-and-dirty* implementations, which are only meant to be a proof-of-concept of the researcher's theoretical work. Typically, this implies a lack of extensibility and comparability as the authors did not have time to bother for a clear, extensible design. We are only aware of two learning libraries that aim for the objectives mentioned above; note that `Java PathFinder` (cf. [4]) also contains a learning submodule (implementing Angluin's L^* algorithm), but this software seems to be too restricted for most cases.

The `LearnLib` library [5] allows learning of deterministic finite-state automata. It is available as a dedicated, password-protected server located at the University of Dortmund and can be accessed via the Internet. The `LearnLib` implements Angluin's L^* algorithm for inferring DFA and some slight variants for deriving Mealy machines.

The *Rich Automata Learning and Testing* library [6] (`RALT`) has been developed in `Java` yielding a platform independent solution. It also implements L^* and three relatives for inferring Mealy machines. Regrettably, `RALT` seems not publicly available.

However, two requirements that seem to be crucial for many user application are clearly missing: Firstly, both libraries are limited to learning Mealy machines in an Angluin setting, but in many environments different learning settings occur. Beyond that, a way to augment the libraries with new learning algorithms, in particular for additional kinds of automata models, is clearly missing. Secondly, as `LearnLib` can be only accessed remotely and `RALT` is not available, it seems impossible to assess their performance; in fact, we were not able to experimentally evaluate or benchmark `libalf` to neither existing library in any appropriate manner. To the best of our knowledge libalf is currently the only available automata learning library that is competitive and flexible enough for real world applications.

3 A library for learning automata: `libalf`

The `libalf` library is an actively developed and stable open source library[1] for learning and manipulating formal languages; it puts the emphasis on learning deterministic and non-deterministic finite-state machines, but can be easily augmented with new automata classes (for instance, `libalf` already supports learning of visibly one-counter automata). As of today, `libalf` comprises a total of nine learning algorithms, cf. Table 1.

Table 1. Algorithms available in `libalf`.

Online algorithms	Offline algorithms
Angluin's L^* (2 variants)	Biermann (2 variants)
NL^* [7]	RPNI
Kearns / Vazirani	DeLeTe2
Visibly 1-counter automata [8]	

`libalf` consists of a core `C++` library and is complemented by two additional components: `liblangen` (a library to generate random regular languages) and `AMoRE++` (a `C++` automata library, among others featuring the antichains algorithm described in [9]). Although written in `C++`, `libalf` fits seamlessly into diverse environments: it runs on `MS Windows`, `Linux`, and `Mac OS` (in 32- and 64-bit) and features a platform independent `Java` interface (using the Java Native Interface `JNI`). In addition, the so-called *dispatcher* implements a network-based client-server architecture, which allows one to run `libalf` remotely, e.g., on a high-performance machine.

[1] `libalf` is freely available on `http://libalf.informatik.rwth-aachen.de/`.

The key objectives of `libalf` are *high flexibility* and *simple extensibility*. High flexibility, on the one hand, means that `libalf` lets the user easily switch between learning algorithms and information sources (often only by changing a single line of code[2]). This allows one to experiment with different learning techniques, making it possible for the user to choose the algorithm best suited for her setting. Moreover, `libalf`'s visualization and logging facilities enable researchers to gain a deeper understanding of the differences of existing and new algorithms.

Simple extensibility, on the other hand, mainly refers to `libalf`'s structured `C++` class hierarchy, especially the learning algorithms and automata models. That allows developers to easily enrich `libalf` with additional features such as new learning algorithms, advanced automata classes, domain-specific optimizations, etc.

Obviously, developing a flexible and easy-to-use library while preserving high extensibility was one of the implementation's most challenging tasks. A comparison of important learning libraries to `libalf` is given in Table 2.

Table 2. Overview over the most important learning libraries in comparison to `libalf`.

	`libalf`	`LearnLib`	`RALT`
Algorithms	online / offline	online	online
	currently 9	1 (`L*`)	1 (`L*`)
Hypotheses	DFA, NFA, Mealy, visibly one-counter, etc.	DFA, Mealy	DFA, Mealy
Open source	yes	no	n/a
Availability	`C++`, `Java` (`JNI`)	`C++`	`Java`
	source code, binary, dispatcher	via Internet connection only	n / a
Specifics	filters, normalizers, statistics, visualization	filters, statistics, visualization	visualization

Technical details. In `libalf` words $w \in \Sigma^*$ (i.e., *queries*) are represented as lists of symbols, where each symbol is a 32-bit integer. Thus, the maximal size $|\Sigma|$ of an alphabet Σ is 2^{32}. For hypotheses, on the other hand, `libalf` provides generic but simple interfaces such that new automata classes can easily be added. However, the `AMoRE++` library can be used if a more powerful automata library is needed.

`libalf`'s main components are the *learning algorithms* and the so-called *knowledgebase*. The knowledgebase is an efficient storage for language information and collects *queries* and *classifications* thereof; in `libalf` a classification can be any `C++` object, but in most algorithms it is a Boolean value. Using an external storage has the advantage of being independent of the choice of the learning algorithm. So it becomes possible to quickly interchange different learning algorithms or run them (even concurrently) on the basis of the same knowledgebase (i.e. queries are only conducted once and are then available to any learning algorithm). Clearly, this helps the user experiment and decide which algorithm to use in her specific setting.

Additionally, `libalf` features two types of domain-specific optimizations: *filters* and *normalizers*. Filters are a means for reducing the number of queries asked to the teacher. The idea is that in many cases the classification of a query can be decided without consulting the teacher just by applying simple domain-specific knowledge; take, for instance, well-formedness of XML-documents as such a criterion. If a query can already be answered by a filter, it is not passed on to the teacher and the number of queries actually asked to the teacher is reduced. Moreover, filters can be composed by logical connectors (*and*, *or*, *not*).

[2] Visit our website for a `Java` online demo on how to employ `libalf` in a user application.

In contrast, *normalizers* are a means to reduce memory consumption during the learning phase. A normalizer defines a domain-specific equivalence relation $\sim \subseteq \Sigma^* \times \Sigma^*$ over all words and only stores data for one representative of each equivalence class (i.e. data for equivalent queries is only queried and stored once). This does not only reduce the consumed memory, but also the number of queries conducted. By subtyping the respective interface, a user can easily define her own domain-specific optimizations.

Finally, `libalf` comprises auxiliary components to ease application development and debugging: a *logger* (an adjustable logging facility an algorithm can write to), extensive *statistics* and methods to produce GraphViz visualizations. All of `libalf`'s components are designed to be used in a plug-and-play manner and, to this end, no knowledge about the libraries implementation is required.

4 Conclusion

`libalf` is a new, comprehensive open-source learning framework, which is easy to use and extend. It gathers several on- and offline learning techniques. The main features of our library and other approaches described previously are summarized in Table 2.

Our learning library is currently used and extended for inferring CFMs from MSC specifications [10] and for learning attractor sets in infinite games (D. Neider, RWTH Aachen). Moreover, there are requests for using `libalf` for searching through source code to find similar code fragments, so-called clones, (E. Jürgen, TU Munich) and for learning black box systems from log files.

For future work, we plan to augment `libalf` with additional learning algorithms, e.g., learning using homing sequences or Trakhtenbrot's algorithm, and to integrate learning techniques for other important language classes, such as transducers, Büchi automata etc. Another ongoing work puts different learning algorithms in comparison. In this project, we compare different online and offline learning algorithms and evaluate their average time complexity. The results obtained so far look very promising.

References

1. Oliveira, A.L., Silva, J.P.M.: Efficient Algorithms for the Inference of Minimum Size DFAs. Machine Learning **44**(1/2) (2001) 93–119
2. Groce, A., Peled, D., Yannakakis, M.: Adaptive model checking. In: TACAS. Volume 2280 of LNCS., Springer (2002) 357–370
3. Habermehl, P., Vojnar, T.: Regular Model Checking Using Inference of Regular Languages. ENTCS **138**(3) (2005) 21–36
4. Giannakopoulou, D., Pasareanu, C.S.: Interface Generation and Compositional Verification in Java Pathfinder. In: FASE. Volume 5503 of LNCS. (2009) 94–108
5. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: LearnLib: a framework for extrapolating behavioral models. STTT **11**(5) (2009) 393–407
6. Shahbaz, M.: Reverse Engineering Enhanced State Models of Black Box Software Components to Support Integration Testing. PhD thesis, Laboratoire Informat. de Grenoble (2008)
7. Bollig, B., Habermehl, P., Kern, C., Leucker, M.: Angluin-Style Learning of NFA. In: IJCAI 2009, AAAI Press (2009) 1004–1009
8. Neider, D., Löding, C.: Learning Visibly One-Counter Automata in Polynomial Time. Technical Report AIB-2010-02, RWTH Aachen (January 2010)
9. Wulf, M.D., Doyen, L., Henzinger, T.A., Raskin, J.F.: Antichains: A new algorithm for checking universality of finite automata. In: CAV. Volume 4144 of LNCS. (2006) 17–30
10. Bollig, B., Katoen, J.P., Kern, C., Leucker, M.: Learning Communicating Automata from MSCs. IEEE TSE To appear.