flexfringe: a passive automaton learning package

Sicco Verwer
Delft University of Technology
Cyber Security group
Email:s.e.verwer@tudelft.nl

Christian A. Hammerschmidt
University of Luxembourg
SEDAN group, SNT
Email: firstname.lastname@uni.lu

***Abstract*—Finite state models, such as Mealy machines or state charts, are often used to express and specify protocol and software behavior. Consequently, these models are often used in verification, testing, and for assistance in the development and maintenance process. Reverse engineering these models from execution traces and log files, in turn, can accelerate and improve the software development and inform domain experts about the processes actually executed in a system. We present *flexfringe*, an open-source software tool to learn variants of finite state automata from traces using a state-of-the-art evidence-driven state-merging algorithm at its core. We embrace the need for customized models and tailored learning heuristics in different application domains by providing a flexible, extensible interface.**

I. Introduction

State machines are key models for the design and analysis of computer systems [1]. In particular, they are frequently used to specify software and protocol behavior [2]. Learning state machines, i.e., reversing the design process [3], enjoys a lot of interest from the software engineering and formal methods communities. Learned state machines provide insight into complex software systems and their properties can be tested using model checking [4] and testing techniques [1]. In the literature, this approach has been used for learning and analyzing models for different types of complex software systems such as web-services [5], X11 windowing programs [6], communication protocols [7], even malicious ones [8], [9], [10], [11], and Java programs [12]. Last year it proved to be powerful indeed, being a key part of the winning contribution to the rigorous examination of reactive systems (RERS) challenge in 2016 [13].

Compared to other, more classical models of data from machine learning, state machines are often seen as more interpretable and more suitable for the domain of software engineering [14], [15]. In our experience, it is often beneficial to learn variants of state machines, such as timed automata [16], mealy machines [17], or register automata [18], rather than plain state machines. This is impossible without modifications to existing (learning) algorithms and programs. We therefore present *flexfringe*, a tool that brings a *flexible* and *extensible* approach to learning variants of automata using a well-known greedy state-merging method [19].

The remainder of this paper is organized as follows: Section II introduces the tool and the core algorithm, Section III outlines the usage of the command line as well as Python interface. Section IV explains the different control options for the algorithm. Before concluding with a short discussion in Section VII, we discuss related work in Section VI.

II. The Tool

flexfringe and its documentation together with full usage examples is made available as open-source at http://www.automatonlearning.net/flexfringe.

A. Design and Goals

Our tool aims to fill the need for a highly customizable state machine learning tool. It evolved from its precursor, called `DFASAT`, which provided a very competitive implementation of a state-merging algorithm, winning the StaMiNa competition [20], [21], and performing well on software related problems in other competitions like SPiCE [22].

The functionality of our tool is centered around one core algorithm, the evidence-driven blue-fringe state-merging algorithm [19], and its probabilistic variants such as ALERGIA [23]. We have adapted the original algorithm by making it flexible and easy to customize. It can now deal with a wide range of heuristics and model types, including probabilistic and non-probabilistic deterministic finite state machines (P/DFA), deterministic finite state transducers like Mealy and Moore machines, as well as ad-hoc defined automata types like regression automata (RA) [24]. The key contributions of our tool are:

- an efficient implementation of the core state-merging algorithm using a union-find data structure to store and undo merges,
- several search methods for finding better models (including the translation to satisfiability from [21]),
- advanced features of state-merging algorithms such as sinks [21] and data streaming [25],
- many parameters that modify the core functionality and search strategy of *flexfringe*,
- the ability to modify the heuristic and type of model and its visualization by adding a single file to the code, and
- an interactive interface and bindings to Python that allows a user to iterate through and modify the different steps of the algorithm in order to fine-tune a newly defined heuristic.

Our goal is to provide practitioners not only with a convenient and easy to use tool, but help them to build customized models, heuristics, and visualizations without the need of a deep understanding of the core state-merging algorithm.

B. How it works: Flexible State-Merging

We provide a brief discussion of the core algorithm, as outlined in Algorithm 2. The sole purpose of this discussion is to provide enough details to understand custom heuristics and extensions to *flexfringe*. More detailed discussions on state-merging can be found in, e.g., [19], [21]. The algorithm centers around a heuristic search that uses a *consistency check* and a *score function*. For classical models like probabilistic and non-probabilistic automata, heuristics with convergence proofs and correctness guarantees exist in a range of paradigms (e.g. in-the-limit learning or PAC-learning [26]).

The algorithm starts by constructing of a tree-shaped automaton from a set of words with optional labels, called the input sample. It contains all sequences from the input sample, with each sequence element as a directed labeled transition, and states added to the beginning, end, and between each symbol of the sequence. Two samples share a path in the tree if they share a prefix. The state-merging algorithm reduces the size of the automaton iteratively by merging pairs of states in the model, and forcing the result to be deterministic. The choice of the pairs, and the evaluation of a merge is made heuristically: first it is checked whether one state is a *sink* (line 6), if not the *consistency* of the merge is *evaluated* (line 7) and *scored* (line 8), and the highest scoring merge is executed (line 12). If no merge is available, the model is extended (line 15). This process is repeated and stops if no merges with high scores are possible. Optionally a search procedure such as backtracking or best-first search can be used on top of this greedy algorithm in order to find better models. *flexfringe* provides a flexible interface that provides the sink definition, and evaluation and scoring functions. Figure 1 shows a tree-shaped automata, and the same automata after some merges. Note that *flexfringe*'s core functionality is not different for labeled and unlabeled inputs. Only the heuristics and types of models being learned change.

III. Usage

There are two ways of using our tool: via the command line interface together with our utility scripts, and via the Python interface directly or in a Jupyter notebook.

A minimal call from command line, running on the data in batch mode, requires providing the heuristic name and

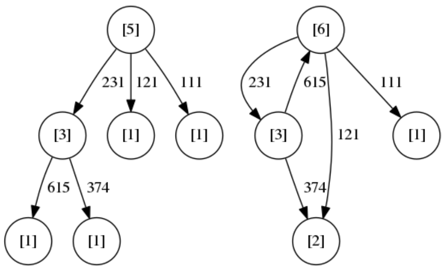

Fig. 1. A prefix tree and a DFA obtained via state-merging. States are circles (annotated with occurrence counts), transitions are directed edges labeled with transition symbol. Starting and ending state are not indicated in this visualization.

```
Require: an input sample $S$
Ensure: $A$ is a DFA that is consistent with $S$
 1: $A$ = apta($S$) {construct the APTA $A$}
 2: $R = \{q_0\}$ {color the start state $q_0$ of $A$ red}
 3: $B = \{q \in Q \setminus R \mid \exists \langle q_0, q, l \rangle \in T\}$
    {color all targets of transitions from $q_0$ children blue}
 4: while $B \neq \emptyset$ do {while $A$ contains blue states}
 5:   for all $b \in B$ and $r \in R$ do
 6:     if sink($b$) is false then
 7:       compute consistency($A, r, b$) of $merge(A, r, b)$
 8:       compute evidence($A, r, b$) of $merge(A, r, b)$
          {find the best performing merge}
 9:     end if
10:   end for
11:   if $\exists b, r$ : evidence($A, r, b$) $\geq$ lowerbound then
12:     $A := merge(A, r, b)$ with highest evidence
        {perform the best merge}
13:     $B := \{q \in Q \setminus R \mid \exists r \in R \wedge \langle r, q, l \rangle \in T\}$
        {recompute the set of blue states}
14:   else
15:     $R := R \cup \{b\}$        // extend the red states, adding $b$
16:     $B := B \cup \{q \in Q \setminus R \mid \exists \langle b, q, l \rangle \in T\}$
17:   end if
18: end while
19: return $A$
```

Fig. 2. Evidence-Driven State-Merging . Lines 6, 7, and 8 contain the *sink*, *consistency*, and *evidence* functions. These are user-defined methods provided by the *flexfringe*'s flexible framework. It can also be set to any of the provided functions such as RPNI [23], ALERGIA [27], MDI [28], EDSM [19], overlap [21], Mealy, or a combination of these. *flexfringe* outputs `m` for a merge (executed in line 12), and `x` for an extend (executed in line 15).

data type as well as the name of the data file. All remaining parameters have default values.

```
$ ./flexfringe --heuristic-name heurstic
   --data-name dtype inputfile.txt
```

The mode and all parameters can be changed via UNIX-style command line options, i.e. with a single - and a letter, or a -- and the full name of the parameter. A list of options is available by calling `./flexfringe --help`. Moreover, we provide the utility script `start.sh` which takes a file listing all the command line options to ease repeated experimentation as follows.

```
$ ./start.sh settings.ini datafile.txt
```

For the Python interface, we exposed the core methods of our tool via `boost.python` and developed wrappers for the core functions. A minimal call consists of creating an estimator object and calling its fit method.

```
import flexfringe as ff
e = ff.DFASATEstimator(
        herustic-name="heuristic",
        data-name="dtype")
e.fit_(dfaFile="inputfile")
```

A. Input Format

The input format is based on the Abbadingo competition format [19]. Each input file starts with a header line containing first the number of sequences in the set and then the number of different symbols occurring across all sequences[1].

Each subsequent line of the corresponds to a single sequence. The first element of the line is a numeric label for the sequence, the second is the number of elements in the sequence. The remaining elements form the sequence, element by element, separated by spaces. The elements can be any sequence of characters without spaces or backslash. Each element can have data attached with a backslash as separator. This data is forwarded to reading functions in the *flexfringe*'s user-defined framework. This makes it possible to learn Mealy machines (data is output) or regression automata (data is a regression target value) by adding only a few lines of code:

```
label length sym1/data1 ... symn/datan
```

Streaming mode, as discussed later, also uses this format, but omits the header. The following example is labeled with 1, has length 4. The events in the sequence are in small caps, and the data added to each element is in capital letters.

```
1 4 on/OK start/RUN stop/STOP off/OK
```

B. Understanding and Using the Output

While running, *flexfringe* outputs a letter as identifier for each action it executes on the input data. There are two basic actions: *extends* `x` and *merges* `m`. Each letter is followed by a number indicating the strength of the evidence for the action; it is the score the heuristic assigned the action, e.g. in this execution are 5 extends and 16 merges.

```
start greedy merging
x47  m21  m21  m13  m12  m10  m9  x12  m19
x23  m19  m9  m8  m5  m3  x20  m13  x14
m7  m6  m5  no more possible merges
```

The output informs a user about the decisions made to obtain the final model and can be very helpful in adjusting parameters. This is particularly helpful when reverse-engineering a model: it is often hard to come up with an error metric to optimize besides the epistemic insight that a human engineer gains from the model. Understanding the algorithmic decisions can help the user to either add more data, tune parameters, or manually exclude decisions considered bad from an engineering point of view. To support this, we developed the *interactive mode* (as discussed below). At each step, the user can choose between all possible actions, e.g.

```
./start-interactive.sh interactive-mealy.ini
          data/original/traces1000.txt
```

gives a list of options and allows the user to choose a particular refinement, or to interactively change a parameter and see the impact on the proposed merges:

```
Possible refinements:
m147, m145, m141, m2, m1, m1, m1, m0, m0, m0,
```

[1]The header is purely for backward and cross-tool compatibility. *flexfringe* automatically extracts this information and does not rely on the header itself.

For instance, changing the search space with `set blueblue 1`, the interactive mode recalculates the merges and gives:

```
Your choice at step 1: set blueblue 1
Possible refinements:
m147, m145, m141, m64, m64, m60, m60, m57, m57,
m46, m46, m43, m43, m41, m41, m40, m40, m36,
m36, m35, m35, m2, m2, m2, m2, m1, m1, m1, m1,
m1, m1, m0, m0, m0, m0, m0, m0, m0, m0, m0, m0,
```

Simultaneously, *flexfringe* outputs the dot files for the last two steps, visualizing the impact of difference choices. In this case, increasing the search space with the `blueblue` option discovers more merges, but the highest scoring merge `m147` was already contained in the smaller search space. See [29] for a discussion and more examples of this mode.

Once finished, *flexfringe* outputs the learned automaton, which are directed graphs with labels for both nodes and edges, as files in Graphviz's dot format. The exact formatting choices for edges and nodes can be modified with a number of `print-*` methods specific to the evaluation function and data classes, which we discuss in Section V.

To use the automaton model to test word acceptance or sample from it, it needs to be parsed. Because *flexfringe* allows for custom models with custom semantics (e.g. guards on transitions), the price for the flexibility is the lack of a general parser. Nevertheless, we provide a short Python script, `evaluate.py`, for plain DFA. The output file also contains the exact command line used to invoke *flexfringe* as well as the git commit id the binary was compiled from to support reproducible experimentation.

IV. Controlling the Algorithm

The core algorithm, beyond the heuristic used in it, is controlled by a series of command lined options. We provide three modes of operation for the command line tool, controlling how the search algorithm is used: batch, stream, and interactive. The mode can be chosen via the `--mode` option.

The conventional way of using our tool is in *batch mode*: A set of collected data is given to the learner, and is processed at once. This has the advantage that all information available is used to learn the model.

In *streaming mode*, *flexfringe* reads lines of samples from standard input and processes them as soon as there is sufficient data to make a decision. The amount of data required is decided using the Hoeffding bound. Similar to streaming decision trees [30], the amount of data required depends on the difference between the best and second-best possible merge. We are still performing a theoretical analysis of this bound. Moreover, we are implementing sketches as in [25] to reduce memory usage. Intermediate models are generated and made available as the stream is processed. This has the advantage of running during data collection, but the disadvantage that it might mis-estimate data and make a wrong decision.

The *interactive mode* supports data exploration, especially if there is only very little data available. Rather than automatically executing the merge heuristic at each step, the current model is displayed and all considerations of the heuristic are outlined. The user now can either follow the suggestion of the merge heuristic, tweak some parameters or even insert more data (if available) and observe the impact of these actions on the merge heuristic's suggestions.

flexfringe has a number of options to control the core state-merging algorithm. Importantly, `--use-sinks` activates the use the *sink* check that excludes states that meet user-defined conditions, e.g., having low frequency counts, having only accepting traces (or none), triggering an error, etc. Excluding pure accepting and rejecting states was used in the Stamina competition winner [21]. Excluding low frequency states is very useful for visualization and debugging purposes (see Figure 3). The options `extent`, `finalred`, `largestblue`, `blueblue`, and `lowerbound` control the search order and search space (i.e., the r and b considered in line 11 of Algorithm 2, and the condition in line 12).

V. Customizing the Heuristics

An important aspect of the design of *flexfringe* is extendability and customization: Providing *flexfringe* with a new merge heuristic and using it should be easy. To implement a custom heuristic, the user needs to be able answer two questions:

- When are two states considered consistent and similar?
- How to quantify similarity of states in a score?

The answers provide the key functionality in the user-provided heuristic: the *consistency check* and the *evidence score* of the heuristic. Both can depend on the sample and the data of the sample. Intuitively, the heuristic tries to generalize the data by identifying pairs of states that have similar future behaviors, i.e., by testing a Markov property. In a probabilistic setting, this similarity is usually be measured by similarity of the empirical probability distributions over the outgoing transition labels. Other common heuristics rely on occurrence information and the identity of symbols, or use global model selection criteria to calculate merge scores.

We provide two classes that encode the required functionality: `evaluation_function` and `evaluation_data`. To build a custom heuristic, the user needs to provide these methods by inheriting from the base class. After recompiling, the custom heuristic can be used by its name using the `heuristic-name` and `data-name` parameters without having to touch any other files. Each node in the initial prefix tree (Algorithm 2, line 1) contains a pointer to an instance of `evaluation_data`, which can contain information of the data in in- or outgoing transitions (or an aggregation thereof). The data is instrumental in learning extended versions of DFA, such as Mealy machines. Moreover, the class also contains `print`-methods to control the style and annotation of states and transitions in the dot files.

VI. Related Work

There are several families of algorithms to learn automata models, each offering different benefits and drawbacks. In [31], the authors provide a C++ implementation of a number of standard algorithms for P/DFA and introduce a noise-resistant heuristic. The paper also contains a short comparison of other

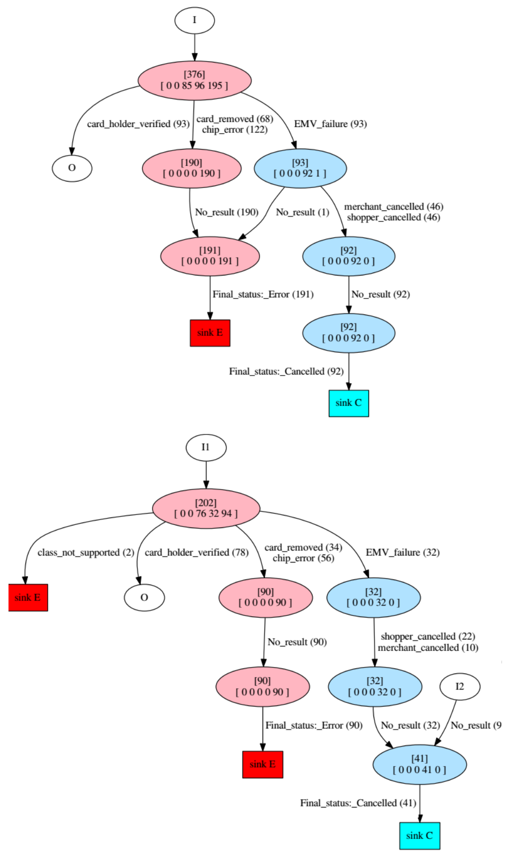

Fig. 3. Two automata parts (top, bottom) comparing different behaviors of card types during payments on a pin entry device with the same firmware, learned from a random sample of 5000 transactions. For details see our companion industry track paper [36]. Using a default heuristic, such automata are typically learned within a few seconds.

tools. In an *active learning* setting, there is *LearnLib* [32]. In a *passive learning* setting (which our tool also falls into), there is a Python package for learning weighted automata with spectral learning approach [33]. There are a number of tool implementing DFA inference for software, e.g. Synoptic [34] for inference over log-files, and InvariMint [35]. While the latter can infer DFA with variable bindings on the transitions, it does not offer the full flexibility *flexfringe* offers.

VII. Future Work and Conclusion

We present *flexfringe*, a flexible state-merging algorithm for passively learning state machine models from trace data. The core algorithm is efficient and mature. We have applied it ourselves to a range of applications such as software bug discovery [36], modeling network traffic [37], and time series regression [24]. Now is the time to collect user feedback and improve the user experience, including the visualization and customization capabilities. To this aim, we publish *flexfringe* as an open source tool and ask the software engineering community to give it a try. Two important future improvements are the use of sketches to learn from high volume/velocity streams, and the ability to provide new heuristics via Python classes rather than C++ classes that require recompilation.

References

[1] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, Aug. 1996.

[2] F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme, *Modeling Software with Finite State Machines: A Practical Approach*. CRC Press, May 2006.

[3] J. E. Cook and A. L. Wolf, "Discovering Models of Software Processes from Event-based Data," *ACM Trans. Softw. Eng. Methodol.*, vol. 7, no. 3, pp. 215–249, Jul. 1998.

[4] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999, google-Books-ID: Nmc4wEaLXFEC.

[5] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli, "Automatic Synthesis of Behavior Protocols for Composable Web-services," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 141–150.

[6] G. Ammons, R. Bodk, and J. R. Larus, "Mining Specifications," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '02. New York, NY, USA: ACM, 2002, pp. 4–16.

[7] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana, "HVLearn: Automated Black-box Analysis of Hostname Verification in SSL/TLS Implementations," in *Proceedings of the 38th IEEE Symposium on Security & Privacy (San Jose, CA*, 2017.

[8] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic Protocol Reverse Engineering from Network Traces." in *USENIX Security Symposium*, 2007, pp. 1–14.

[9] C. Y. Cho, D. Babi , E. C. R. Shin, and D. Song, "Inference and Analysis of Formal Models of Botnet Command and Control Protocols," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 426–439.

[10] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol Specification Extraction," in *2009 30th IEEE Symposium on Security and Privacy*, May 2009, pp. 110–125.

[11] G. Pellegrino, Q. Lin, C. Hammerschmidt, and S. E. Verwer, "Learning Behavioral Fingerprints From Netflows Using Timed Automata," in *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*, May 2017.

[12] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin, "Reverse Engineering State Machines by Interactive Grammar Inference," in *14th Working Conference on Reverse Engineering (WCRE 2007)*, Oct. 2007, pp. 209–218.

[13] R. Smetsers, J. Moerman, M. Janssen, and S. Verwer, "Complementing Model Learning with Mutation-Based Fuzzing," *arXiv:1611.02429 [cs]*, Nov. 2016, arXiv: 1611.02429.

[14] C. A. Hammerschmidt, Q. Lin, S. Verwer, and R. State, "Interpreting Finite Automata for Sequential Data," *arXiv:1611.07100 [cs, stat]*, Nov. 2016, arXiv: 1611.07100.

[15] F. Vaandrager, "Model Learning," *Commun. ACM*, vol. 60, no. 2, pp. 86–95, Jan. 2017.

[16] S. Verwer, C. Witteveen, M. De Weerdt, TU Delft: Electrical Engineering, Mathematics and Computer Science: Software Technology, and TU Delft, Delft University of Technology, "Efficient Identification of Timed Automata: Theory and practice," Mar. 2010.

[17] J. De Ruiter and E. Poll, "Protocol State Fuzzing of TLS Implementations." in *USENIX Security Symposium*, 2015, pp. 193–206.

[18] F. Aarts, F. Howar, H. Kuppens, and F. Vaandrager, "Algorithms for Inferring Register Automata," in *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Springer, Berlin, Heidelberg, Oct. 2014, pp. 202–219.

[19] K. Lang, "Evidence driven state merging with search," 1998.

[20] N. Walkinshaw, B. Lambeau, C. Damas, K. Bogdanov, and P. Dupont, "STAMINA: a competition to encourage the development and assessment of software model inference techniques," *Empirical Software Engineering*, vol. 18, no. 4, pp. 791–824, Aug. 2013.

[21] M. J. H. Heule and S. Verwer, "Software model synthesis using satisfiability solvers," *Empirical Software Engineering*, vol. 18, no. 4, pp. 825–856, Aug. 2012.

[22] B. Balle, R. Eyraud, F. M. Luque, A. Quattoni, and S. Verwer, "Results of the sequence prediction challenge (spice): a competition on learning the next symbol in a sequence," in *International Conference on Grammatical Inference*, 2017, pp. 132–136.

[23] J. Oncina and P. Garcia, "Identifying Regular Languages In Polynomial Time," in *Advances in ...* World Scientific, 1992, pp. 99–108.

[24] L. Qin, C. Hammerschmidt, G. Pellegrino, and S. Verwer, "Short-term Time Series Forecasting with Regression Automata," in *MiLeTS Workshop at KDD 2016*, 2016, oCLC: 858090417.

[25] B. Balle, J. Castro, and R. Gavald, "Adaptively learning probabilistic deterministic automata from data streams," *Machine Learning*, vol. 96, no. 1-2, pp. 99–127, Jul. 2014.

[26] J. Heinz and J. M. Sempere, *Topics in Grammatical Inference*. Springer, 2016.

[27] R. C. Carrasco and J. Oncina, "Learning stochastic regular grammars by means of a state merging method," in *Grammatical Inference and Applications*. Springer, Berlin, Heidelberg, Sep. 1994, pp. 139–152.

[28] F. T. Thollard, "Probabilistic DFA Inference using Kullback-Leibler Divergence and Minimality," in *In Seventeenth International Conference on Machine Learning*. Morgan Kauffman, 2000, pp. 975–982.

[29] C. A. Hammerschmidt, R. State, and S. Verwer, "Human in the Loop: Interactive Passive Automata Learning via Evidence-Driven State-Merging Algorithms," *arXiv:1707.09430 [cs, stat]*, Jul. 2017, arXiv: 1707.09430.

[30] P. Domingos and G. Hulten, "Mining high-speed data streams," in *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2000, pp. 71–80.

[31] M. Ortolani, P. Cottone, and G. Pergola, "Gl-learning: An optimized framework for grammatical inference," 2016.

[32] M. Isberner, F. Howar, and B. Steffen, "Learning register automata: from languages to program structures," *Machine Learning*, vol. 96, no. 1-2, pp. 65–98, Jul. 2014.

[33] D. Arrivault, D. Benielli, F. Denis, and R. Eyraud, "Sp2learn: A Toolbox for the spectral learning of weighted automata," in *International Conference on Grammatical Inference*, 2017, pp. 105–119.

[34] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 267–277.

[35] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy, "Unifying FSM-inference algorithms through declarative specification," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 252–261.

[36] R. Wieman, W. Lobbezoo, M. Aniche, S. Verwer, and A. van Deursen, "An Experience Report on Applying Passive Learning in a Large-Scale Payment Company," in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, Sep. 2017.

[37] C. Hammerschmidt, S. Marchal, R. State, G. Pellegrino, and S. Verwer, "Efficient Learning of Communication Profiles from IP Flow Records," in *Local Computer Networks (LCN), 2016 IEEE 41st Conference on*. IEEE, 2016, pp. 559–562.