Journal of Machine Learning Research 1–10

Learning of Procedural Systems: Counterexample Projection and Decomposition

Markus Frohme MARKUS.FROHME@CS.TU-DORTMUND.DE
Bernhard Steffen STEFFEN@CS.TU-DORTMUND.DE
Chair of Programming Systems, Faculty of Computer Science, TU Dortmund, Germany.

Abstract

Our recently developed active automata learning algorithm for systems of procedural automata (SPAs) splits the inference of a context-free/procedural system into a simultaneous inference of individual DFAs for each of the involved procedures. This paper concretizes two essential features of our algorithm: counterexample projection and counterexample decomposition, which can be considered the technical key for achieving a modular learning algorithm of high efficiency.

Keywords: active automata learning, procedural systems, context-free languages

1. Introduction

Our recently developed active automata learning algorithm for systems of procedural automata (SPAs) Frohme and Steffen (2018, 2019) splits the inference of a context-free/procedural system into a simultaneous inference of individual DFAs for each of the involved procedures. Key to our approach are the two concepts of projection and expansion: global counterexample traces are transformed into counterexamples for the DFAs of concerned procedures, and membership queries for the individual DFAs are expanded to membership queries of the global system.

This paper focuses on the SPA counterexample decomposition and the subsequent counterexample projection step, as the sketch of the simpler query expansion step provided in Frohme and Steffen (2018) is sufficiently detailed. Figure 1 illustrates three essential characteristics of SPAs, which are important to understand our learning approach.

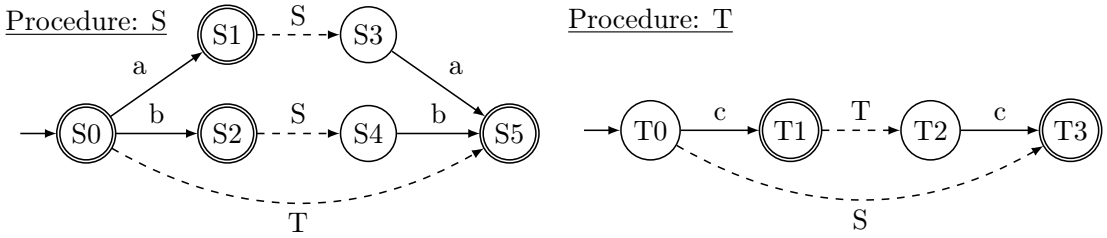

Figure 1: An SPA representation of the context-free grammar of Figure 2 (left).

- the intuitive structure: the operational semantics of SPAs follow the copy-rule semantics, i.e. upon encountering a procedural call the control is transferred to the respective procedural automaton from which it only can return at specific states. While this is a universal concept that is independent of the automaton type of the procedures and thus can be realized on a

purely syntactical level (e.g. via graph transformation/rewriting Rensink (2004)), we focus on systems modeled in terms context-free grammars, which semantically associate an *accepted word* with a successful run of a system.

In order to describe context-free systems via DFAs, we assume the procedural calls as observable, which we justify by the fact that in practice, the required observability can be achieved via easy instrumentation Frohme and Steffen (2018). When ignoring this control overhead, the set of accepting runs coincides with the context-free language corresponding to the procedural system.

- the expressive power: SPAs cover the full spectrum of context-free languages. E.g., the SPA shown in Figure 1 "implements" the language of all palindromes over the alphabet $\{a, b, c\}$.
- the role of procedural names (non-terminals): they can be considered as 'architectural knowledge' about the system to be learned.[1] In this example it imposes a (here intended, but from the mere language perspective unnecessary) separate treatment of symbol c, something which could not be observed simply on the basis of terminal words. This allows one to represent the compositional architecture of the system in terms of intuitive models.

Outline

We continue in Section 2 with introducing preliminary terminology and concepts from Frohme and Steffen (2019) and related fields of research. Section 3 presents our main contribution of the paper: Concretizing the two essential steps of counterexample projection and counterexample decomposition. Section 4 concludes the paper and provides some directions for future work.

2. Preliminaries and related work

The formal foundation of our learning approach, similar to many other active automata learning algorithms, is the minimal adequate teacher (MAT) framework proposed by Angluin Angluin (1987). Key to this framework is the existence of a *teacher* that is able to answer *membership queries*, i.e. questions, whether a word is a member of the target language, and *equivalence queries*, i.e. questions, whether a tentative hypothesis exactly recognizes the target language. Active automata learning (for regular languages) can then be summarized as the iterative process of exploration and verification to discover the equivalence classes of the Myhill-Nerode congruence Nerode (1958) for the target language. We expect the reader to be familiar with the general process and formalities of active automata learning. For a more thorough introduction (to the regular case) see e.g. Steffen et al. (2011) or (Kearns and Vazirani, 1994, Chapter 8).

Regular languages are not powerful enough to capture the key characteristics of procedural systems, which support mutually recursive calls between their sub-procedures and thereby allow to express the entire spectrum of context-free languages. These semantics are, however, expressible with context-free languages. Angluin herself already reasoned about the inference of context-free languages Angluin (1987), but her extensions required for answering e.g. membership queries have – at least to the knowledge of the authors – prevented any practical application.

For inferring context-free/procedural systems, we propose an instrumentation similar to the idea of parenthesis grammars McNaughton (1967): Each invocation of a procedure `P` can be observed by means of a *call* symbol `P'`, which denotes the start of a specific

1. Exploiting given (perhaps architectural) knowledge about the system to be learned is one of the most promising approaches to boost automata learning for large-scale practical application.

procedure, as well as a *return* symbol `R`, which denotes its termination. An example of this instrumentation is given in Figure 2 (right).

```
S -> a |
     a S a |
     b |
     b S b |
     T |
     ε
T -> c |
     c T c |
     S
```

```
S -> S' a R |
     S' a S a R |
     S' b R |
     S' b S b R |
     S' T R |
     S' R
T -> T' c R |
     T' c T c R |
     T' S R
```

Figure 2: On the left: production rules of an exemplary context-free grammar for palindromes over the three terminal symbols `a`, `b`, `c`, using two non-terminal symbols `S` and `T`. On the right: production rules of the instrumented system, introducing two new (observable) *call* symbols `S'` and `T'` and an (observable) *return* symbol `R`.

For software-based systems, this instrumentation can easily be integrated via, e.g., aspect-oriented programming or proxying in object-oriented programming. In certain application domains – especially tag languages such as (DTD-based) XML – these observable entry- and exit-points require no instrumentation at all.

The idea of assigning specific semantics to certain input symbols is conceptually related to *visibly pushdown languages* (VPLs) Alur and Madhusudan (2004); Alur et al. (2005). In fact, our instrumented systems can be described by a *visibly pushdown automaton* (VPA). However, VPAs operate on a single global state space, whereas our compositional SPA approach allows for truly independent procedures, which proves beneficial during our learning process and counterexample decomposition. For a more involved discussion of the difference between the two formalisms, see Frohme and Steffen (2019). We continue to introduce the formal definitions and notations we use throughout the paper:

Definition 1 (SPA alphabet) *An* SPA alphabet $\Sigma = \Sigma_c \uplus \Sigma_i \uplus \{R\}$ *is the disjoint union of three finite sets, where* Σ_c *denotes the* call *alphabet,* Σ_i *denotes the* internal *alphabet and* R *denotes the* return *symbol.*

An SPA alphabet can be seen as a special case of a *visibly pushdown alphabet* Alur and Madhusudan (2004); Alur et al. (2005). We choose a distinct name here in order to address the specifics of SPA alphabets like having a shared return symbol for all procedures. In our palindrome example in Figure 2 (right), the alphabet definition would be $\Sigma = \{S', T'\} \uplus \{a, b, c\} \uplus \{R\}$. Note, that for reasons of simplicity, we will continue to use a single identifier for both the observable symbol and the corresponding procedure if no distinction between the two is required.

An SPA word w is then defined as a sequence of symbols over Σ (written as $w \in \Sigma^*$). For $1 \leq i \leq j \leq |w|$, we write $w[i, j]$ to denote the sub-sequence of w starting at the symbol at position i and ending at position j (inclusive). For any $i > j$, $w[i, j]$ denotes the empty word ε. Furthermore, we limit ourselves to only consider *well-matched* words as only those have the chance to be accepted. Intuitively, a well-matched word is a word, where every call symbol is succeeded (at some point) by a matching return symbol and no unmatched

call or return symbols exist. To formalize this constraint, let us further introduce the idea of call and return balance, which will also be used later for counterexample decomposition.

Definition 2 (Call/Return balance) *Let $\Sigma = \Sigma_c \uplus \Sigma_i \uplus \{R\}$ be an SPA alphabet. The call/return balance is a function $\beta\colon \Sigma^* \to \mathbb{Z}$, defined as*

$$\beta(\varepsilon) = 0$$
$$\beta(u \cdot v) = \beta(u) + \begin{cases} 1 & \text{if } v \in \Sigma_c \\ 0 & \text{if } v \in \Sigma_i \\ -1 & \text{if } v = R \end{cases} \qquad \forall u \in \Sigma^*, v \in \Sigma$$

Given the above definition of the call/return balance, we can formally define well-matched words as words $w \in \Sigma^*$, where every prefix u satisfies $\beta(u) \geq 0$ and every suffix v satisfies $\beta(v) \leq 0$.

2.1. Systems of procedural automata

Definition 3 (Procedural automaton) *Let $\Sigma = \Sigma_c \uplus \Sigma_i \uplus \{R\}$ be an SPA alphabet and $c_i \in \Sigma_c$ denote a procedure. A procedural automaton for procedure c_i over Σ is a deterministic finite automaton $P^{c_i} = (Q^{c_i}, q_0^{c_i}, \delta^{c_i}, Q_F^{c_i})$, where*

- *Q^{c_i} denotes the finite, non-empty set of states,*
- *$q_0^{c_i} \in Q^{c_i}$ denotes the initial state,*
- *$\delta^{c_i} : Q^{c_i} \times (\Sigma_c \cup \Sigma_i) \to Q^{c_i}$ denotes the transition function, and*
- *$Q_F^{c_i} \subseteq Q^{c_i}$ denotes the set of accepting states.*

We define $L(P^{c_i})$ as the language of P^{c_i}, i.e. the set of all accepted words of P^{c_i}.

Definition 4 (System of procedural automata) *Let $\Sigma = \Sigma_c \uplus \Sigma_i \uplus \{R\}$ be an SPA alphabet and P^{c_i} denote a procedural automaton for a corresponding $c_i \in \Sigma_c$. A* system *of procedural automata S is given by the tuple $(P^{c_1}, ..., P^{c_q})$, with $q = |\Sigma_c|$.*

In essence, a procedural automaton resembles a deterministic finite automaton (DFA) that accepts the language of right-hand sides of production-rules for a specific non-terminal, where (observable) call symbols represent the corresponding non-terminals. Consequently, a *system* of procedural automata (with a designated procedure $c^* \in \Sigma_c$ marked as *initial* procedure) constitutes a context-free grammar. Both formalisms are equally expressive, as one can easily transform one into the other (a CFG-to-SPA transformation is trivial, for the SPA-to-(instrumented-)CFG transformation see Definitions 7 and 8 in Appendix A). An example of the SPA representation of the context-free grammar of Figure 2 is shown in Figure 1.

For defining the global acceptance criterion of an SPA, we assume that the reader is familiar with the commonly known properties of CFGs – especially the induced language $L(G)$ of a context-free grammar G (Hopcroft et al., 2001, Chapter 5).

Definition 5 ((instrumented) acceptance criterion of an SPA) *Let $S = (P^{c_1}, ..., P^{c_q})$ be an SPA, $c^* \in \Sigma_c$ a designated start procedure and G be the induced (instrumented) context-free grammar of S. The SPA S accepts a word $w \in \Sigma^*$ iff $w \in L(G)$, i.e. $L(S) = L(G)$.*

3. Global and local counterexamples

SPAs are characterized by their individual procedures. Therefore, the main task of inferring an SPA is to infer each of the individual procedural automata, which on its own is a regular inference problem. Our SPA learner coordinates regular learning algorithms (one for each procedure) and transforms information between the local procedural learners and the global (instrumented) context-free system under learning (SUL). Key to our approach is a translation layer that bridges between the view of the entire system and the local view concerning the individual procedural automata: *Local queries* of the procedural learning algorithms are expanded to *global queries* of the instrumented SUL, and *global counterexample traces* for the global system are projected onto *local counterexample traces* of the concerned procedural automata.

For being able to perform these translations we maintain so-called *access*, *terminating* and *return* sequences. Intuitively, these sequences store information about how a procedural automaton can be accessed within the global SPA, how a successfully terminating run of a procedure looks like, and how global termination can be achieved after executing a procedure (accessed by the matching access sequence). As the membership query expansion has been described in sufficient detail in Frohme and Steffen (2018) we focus here on the counterexample projection and the counterexample decomposition. Although, chronologically, counterexample decomposition precedes counterexample projection, we present the counterexample projection first, as projection is also applied during the query translation required for the counterexample decomposition.

3.1. Counterexample projection

Global counterexamples need to be translated to local counterexamples in order to allow the regular learner to refine the corresponding local hypothesis. During counterexample decomposition (see Section 3.2 for details), one obtains a decomposition $(u, a, v) \in \Sigma^* \times (\Sigma_c \cup \Sigma_i) \times \Sigma^*$, such that after parsing u, the input a identifies a wrong transition in the tentative hypothesis, which can be witnessed by the distinguishing suffix v. The corresponding projection step, which essentially reverses query expansion, is a bit more involved. Given a global counterexample ce and the position $1 \leq l \leq |ce|$ of a, at which the information of interest is located, we first extract the well-matched subword surrounding said position. We determine l_{min}, l_{max} as follows:

$$l_{min} = \max_{i \in \{0,\dots,l-1\}} i+1 \qquad s.t.\ \beta(ce[1,i]) < \beta(ce[1,l-1])$$

$$l_{max} = \min_{i \in \{l,\dots,|ce|\}} i \qquad s.t.\ \beta(ce[1,i]) < \beta(ce[1,l-1])$$

Intuitively, l_{min} marks the index of the call symbol of the procedure that performs the violating transition, whereas l_{max} marks the index of the return symbol terminating the procedure in question. After computing the two limits, we can extract from $ce[l_{min}]$ the procedure that needs refinement, and from $ce[l_{min}+1, l_{max}-1]$ the relevant input sequence for the local learner. This input sequence, however, still needs to be translated into a local context. This is done by symbol-wise processing, which leaves internal symbols unchanged and whenever a call (return) symbol is encountered, it removes all following (previous) symbols until the matching return (call) symbol is encountered. See Figure 3 for illustration.

As indicated, the decomposition of the counterexample can be transferred directly from the global context (g) to the local context (l) of the concerned procedure to trigger the local refinement. The constraint of $a \in \Sigma_c \cup \Sigma_i$ is addressed in more detail in Section 3.2.

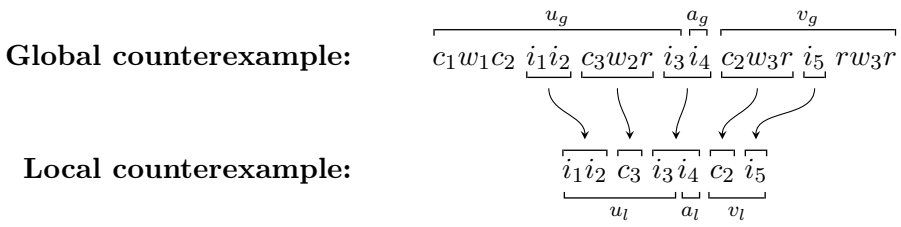

Figure 3: The projection of a global counterexample (in which the violating transition occurs when parsing i_4) to a local counterexample for the procedural automaton of c_2.

3.2. Counterexample decomposition

The following discussion is based on the work of Rivest & Schapire Rivest and Schapire (1989), who presented this type of decomposition of counterexamples for regular systems and the work of Isberner (Isberner, 2015, Chapter 6), who has lifted these results to the context of visibly pushdown languages. Intuitively, counterexample decomposition proceeds by tracing the input sequence of the given counterexample both in the system under learning and in the current hypothesis, to identify an input symbol, which transitions the SUL and the current hypothesis into (provable) different successor states. This process is sketched in Figure 4:

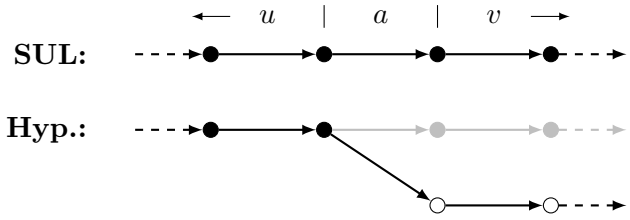

Figure 4: The decomposition of a (regular) counterexample into a tuple $(u, a, v) \in \Sigma^*_{reg} \times \Sigma_{reg} \times \Sigma^*_{reg}$, such that a denotes an input that transitions the current hypothesis into a (provable) different state compared to the SUL.

In the regular case, i.e. for a regular language over some finite alphabet Σ_{reg}, the process of analyzing a counterexample $ce \in \Sigma^+_{reg}$ is given by finding a decomposition $(u, a, v) \in \Sigma^*_{reg} \times \Sigma_{reg} \times \Sigma^*_{reg}$ such that $ce = uav$ and $mq([u]_{\mathcal{H}} av) \neq mq([ua]_{\mathcal{H}} v)$. Here, mq returns the answer to a given membership query and $[u]_{\mathcal{H}}$ returns the access sequence of the hypothesis state reached by u, i.e. the unique representative of the associated equivalence class (cf. Section 2). For such a decomposition, after reading u, a transitions the SUL and the hypothesis into different states (reached by $[u]_{\mathcal{H}} a$ and $[ua]_{\mathcal{H}}$), which can be proven by appending the distinguishing suffix v.

In the context of our instrumented context-free languages, special notice needs be given to the call and return symbols, because they reliably synchronize the two systems as they isolate individual sub-procedures. This allows us to extract individual procedures for hypothesis refinements. An example of this synchronization is given in Figure 5:

Without loss of generality, Figure 5 shows the scenario for a positive counterexample, i.e. a word that is accepted by the SUL but rejected by the conjectured hypothesis SPA, because

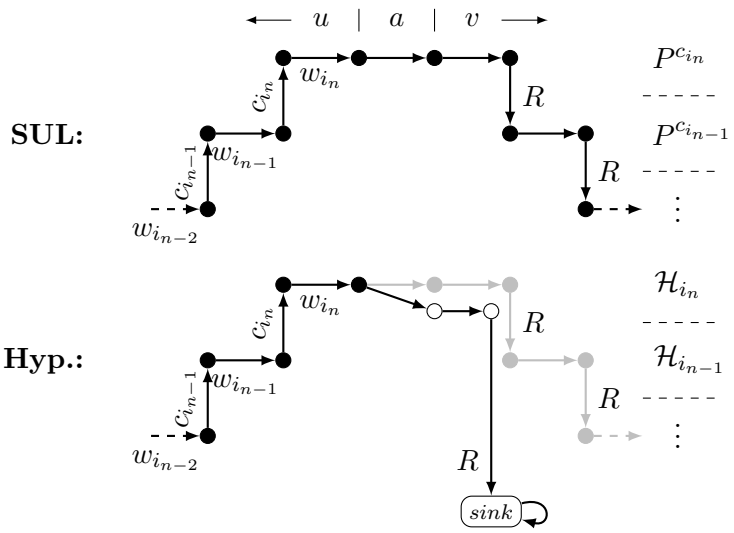

Figure 5: The decomposition of an instrumented counterexample into a tuple $(u, a, v) \in \Sigma^* \times (\Sigma_c \cup \Sigma_i) \times \Sigma^*$, where the access sequence u consists of well-matched subwords $w_i \in \Sigma^*$ and unmatched call symbols $c_i \in \Sigma_c$.

a return symbol is encountered in a non-accepting state of hypothesis $\mathcal{H}_{i_n}$. Equivalently, a negative counterexample, i.e. a word that is rejected by the SUL but accepted by the conjectured hypothesis SPA, would cause such a sink-transition in the system under learning.

For the computation of access sequences ($[u]_{\mathcal{H}}$), we have to account for the special role of call and return symbols: For $u = c_{i_1} w_1 \cdots c_{i_n} w_n$, where $w_i \in \Sigma^*$ represent well-matched words and $c_i \in \Sigma_c$ represent (unmatched) call symbols, $[u]_{\mathcal{H}}$ resolves to $c_{i_1}[w_1]_{\mathcal{H}_{i_1}} \cdots c_{i_n}[w_n]_{\mathcal{H}_{i_n}}$, where $\mathcal{H}_i$ represents the tentative hypothesis of procedure c_i. This modification allows us to determine the access sequence of a (nested) procedural hypothesis state similar to the regular case, while at the same preserving the structural/hierarchical information of the unmatched call symbols, which guarantees that queries in the form of $[u]_{\mathcal{H}} av$ (or $[ua]_{\mathcal{H}} v$) are still well-matched.

During this kind of counterexample decomposition, we make heavy use of our query translation layer: The well-matched subwords w_j may contain arbitrarily nested procedural calls. For determining the access sequences in the local hypotheses $\mathcal{H}_{i_j}$, we project each (nested) invocation to a single call symbol (cf. Figure 3) to trace the corresponding transitions in $\mathcal{H}_{i_j}$. Similarly, if an extracted access sequence $[w_j]_{\mathcal{H}_{i_j}}$ contains call symbols, we expand them with their corresponding terminating sequence prior to contacting the membership oracle to ensure the well-matchedness of the query.

Special notice should be given to the case where we analyze a decomposition with $a = R$. In this situation, the two sequences $[u]_{\mathcal{H}} av$ and $[ua]_{\mathcal{H}} v$ are given by $c_{i_1}[w_1]_{\mathcal{H}_{i_1}} \cdots c_{i_n}[w_n]_{\mathcal{H}_{i_n}} Rv$ and $c_{i_1}[w_1]_{\mathcal{H}_{i_1}} \cdots c_{i_{n-1}}[w_{n-1} c_{i_n} w_n R]_{\mathcal{H}_{i_{n-1}}} v$, respectively. The return action may expose faulty behavior because either $\mathcal{H}_{i_n}$ wrongfully parses (the translated) w_n, or $\mathcal{H}_{i_{n-1}}$ wrongfully parses (the translated) $w_{n-1} c_{i_n} w_n R$, so we need to look at two hypotheses. If we analyze a positive counterexample, we know that both $c_{i_{n-1}}$ and c_{i_n} must accept the (translated) sequences and we can simply select the hypothesis that rejects its respective (translated) sequence. If we analyze a negative counterexample, we need to consider two situations, because $mq([u]_{\mathcal{H}} av) \neq mq([ua]_{\mathcal{H}} v)$ only gives the information of a mismatch

location: If $[u]_{\mathcal{H}}av$ is rejected and $[ua]_{\mathcal{H}}v$ is accepted, we know the error is located in $\mathcal{H}_{i_{n-1}}$ because $\mathcal{H}_{i_n}$ correctly rejected w_n. In the other case ($[u]_{\mathcal{H}}av$ is accepted and $[ua]_{\mathcal{H}}v$ is rejected), we know the error is located in $\mathcal{H}_{i_n}$ because $\mathcal{H}_{i_{n-1}}$ correctly rejected (the translated) $w_{n-1}c_{i_n}w_nR$. Once we have identified the faulty hypothesis, we can continue to analyze the counterexample within the range of the procedural trace and ensure that for the final decomposition we have $a \in \Sigma_c \cup \Sigma_i$. Luckily, distinguishing the two cases does not require additional queries, as the answers to $mq([u]_{\mathcal{H}}av)$ and $mq([ua]_{\mathcal{H}}v)$ are already known from determining a and the continued counterexample decomposition does not impact the asymptotic query complexity of the counterexample decomposition.

Please note, that these translations have a lot of potential for improving query performance, because for the expansion steps one can exchange access sequences, terminating sequences, and return sequences, whenever one has found shorter candidates during the learning process. This interchangeability is only possible, because the individual procedures in our SPA approach are truly independent (as they resemble a context-*free* grammars). Comparable formalisms, such as VPAs, need to parse the complete counterexample as-is, because every single symbol impacts the globally shared state of a VPA.

Summing up, the refinement of a tentative SPA hypothesis consists of the following three steps:

1. By using our global acceptance criterion (cf. Definition 5) and the concept of query translation, we can pinpoint a single symbol in the global counterexample that transitions a procedural hypothesis into a wrong state (counterexample decomposition).
2. This symbol allows one to extract the affected procedure that performs said transition ($ce[l_{min}]$, cf. Section 2).
3. Using counterexample projection, the projected counterexample can be passed to the local learning algorithm of the identified procedure to initiate the necessary local refinement process.

The correctness of this approach is proved in Frohme and Steffen (2019):

Theorem 6 (Correctness and termination) *Having access to a MAT teacher for an instrumented context-free language L, our learning algorithm determines a canonical SPA $S = (P^{c_1}, ..., P^{c_q})$ for L requiring at most $n + 1$ equivalence queries, where $n = \sum_{c_i \in \Sigma_c} |Q^{c_i}|$.*

4. Conclusion and future work

In this paper, we have concretized two essential features of our SPA learning algorithm: counterexample projection and counterexample decomposition, which has only been sketched in Frohme and Steffen (2018), and which can be considered the technical key for achieving a modular learning algorithm of high efficiency. In addition to allowing a modular treatment of the involved procedural automata, which keeps the complexity of the algorithm at the level of regular learning, the involved query translation, which bridges between a global SPA view and a local view concerning the individual procedural systems, provides a high potential of dynamic query optimization: One can exchange access sequences, terminating sequences, and return sequences whenever one has found shorter candidates during the learning process. This interchangeability is only possible, because the individual procedures in our SPA approach are truly independent, in contrast to, e.g., a VPA representation.

Currently, we are investigating the power of SPAs for "never-stop learning" Bertolino et al. (2012) which has the potential to particularly profit form dynamic query optimizations. Our first experimental results concerning counterexample lengths of 100.000 and more are very promising.

References

Rajeev Alur and Parthasarathy Madhusudan. Visibly pushdown languages. In *Proceedings of the 36th annual ACM symposium on Theory of computing*, pages 202–211. ACM, 2004.

Rajeev Alur, Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. *Congruences for Visibly Pushdown Languages*, pages 1102–1114. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-31691-6. doi: 10.1007/11523468_89. URL https://doi.org/10.1007/11523468_89.

Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.

Antonia Bertolino, Antonello Calabrò, Maik Merten, and Bernhard Steffen. Never-stop Learning: Continuous Validation of Learned Models for Evolving Systems through Monitoring. *ERCIM News*, 2012(88):28–29, 2012. URL http://ercim-news.ercim.eu/en88/special/never-stop-learning-continuous-validation-of-learned-models-for-evolving-systems-through-monitori

Markus Frohme and Bernhard Steffen. Active Mining of Document Type Definitions. In Falk Howar and Jiři Barnat, editors, *23rd International Conference, FMICS 2018, Maynooth, Ireland, September 3-4, 2018, Proceedings*. Springer International Publishing, 2018. ISBN 978-3-030-00243-5.

Markus Frohme and Bernhard Steffen. Compositional Learning of Mutually Recursive Procedural Systems. 2019. Under submission.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - (2. ed.)*. Addison-Wesley series in computer science. Addison-Wesley-Longman, 2001. ISBN 978-0-201-44124-6.

Malte Isberner. *Foundations of Active Automata Learning: An Algorithmic Perspective*. PhD thesis, Technical University Dortmund, Germany, 2015. URL http://hdl.handle.net/2003/34282.

Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-11193-4.

Robert McNaughton. Parenthesis grammars. *J. ACM*, 14(3):490–500, July 1967. ISSN 0004-5411. doi: 10.1145/321406.321411. URL http://doi.acm.org/10.1145/321406.321411.

A. Nerode. Linear Automaton Transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958. ISSN 00029939.

Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, pages 479–485, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-25959-6.

Ronald L. Rivest and Robert E. Schapire. Inference of Finite Automata Using Homing Sequences. In *Proc. 21^{st} ACM Symp. on Theory of Computing*, pages 411–420. MIT Laboratory for Computer Science, ACM Press, May 1989. doi: http://doi.acm.org/10.1145/73007.73047.

Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to Active Automata Learning from a Practical Perspective. In Marco Bernardo and Valérie Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-21455-4_8.

Appendix A. Induced CFGs of an SPA

For formalizing the transformation of an SPA into a context-free grammar, we use grammars in the form of $G = (N, T, P, S)$ (N = non-terminals, T = terminals, $P \subseteq N \times (N \cup T)^* =$ production relation, S = start non-terminal). Note, that specifications of context-free grammars in extended BNF allow to specify the right-hand sides of production rules as regular expressions.

Definition 7 (Induced context-free grammar of an SPA) *Let $\Sigma = \Sigma_c \uplus \Sigma_i \uplus \{R\}$ be an SPA alphabet, $c^* \in \Sigma_c$ a designated start procedure and $S = (P^{c_1}, ..., P^{c_q})$ a corresponding SPA. Furthermore, let reg be a function, that returns for every regular language L_{reg} a regular expression exactly describing the language, i.e. $L_{reg} = L(reg(L_{reg}))$ (Hopcroft et al., 2001, Chapter 3). Using S, one constructs a context-free grammar $G = (N, T, P, S)$ in EBNF, such that:*

- $N = \Sigma_c$
- $T = \Sigma_i$
- $P = \{(c_i, reg(L(P^{c_i}))) \mid c_i \in \Sigma_c\}$
- $S = c^*$

Note here, that the observable call symbols are used as non-terminals and thus are not part of the induced language. This is because these symbols were only introduced as part of our instrumentation process. With this construction of the (original) system, one can easily see, how the instrumented system can be constructed (cf. Figure 2: left vs. right).

Definition 8 (Induced instrumented context-free grammar of an SPA) *Let $\Sigma = \Sigma_c \uplus \Sigma_i \uplus \{R\}$ be an SPA alphabet, $c^* \in \Sigma_c$ a designated start-procedure and $S = (P^{c_1}, ..., P^{c_q})$ a corresponding SPA. Furthermore, let reg be defined as in Definition 7 and nt be a function, that returns for every regular expression over Σ the same expression, where every observable call symbol $c_i \in \Sigma_c$ has been replaced with an arbitrary but unique non-terminal symbol $x_i \notin \Sigma$. Using S, one constructs a context-free grammar $G = (N, T, P, S)$ in EBNF, such that:*

- $N = \{nt(c_i) \mid c_i \in \Sigma_c\}$
- $T = \Sigma_c \cup \Sigma_i \cup \{R\}$
- $P = \{(nt(c_i), c_i \cdot nt(reg(L(P^{c_i}))) \cdot R) \mid c_i \in \Sigma_c\}$
- $S = nt(c^*)$