International Journal on Software Tools for Technology Transfer (2021) 23:521–543
https://doi.org/10.1007/s10009-021-00634-y

GENERAL

Regular

Compositional learning of mutually recursive procedural systems

Markus Frohme[1] · **Bernhard Steffen**[1]

Published online: 5 October 2021

Abstract
This paper presents a compositional approach to active automata learning of Systems of Procedural Automata (SPAs), an extension of Deterministic Finite Automata (DFAs) to systems of DFAs that can mutually call each other. SPAs are of high practical relevance, as they allow one to efficiently learn intuitive recursive models of recursive programs after an easy instrumentation that makes calls and returns observable. Key to our approach is the simultaneous inference of individual DFAs for each of the involved procedures via expansion and projection: membership queries for the individual DFAs are expanded to membership queries of the entire SPA, and global counterexample traces are transformed into counterexamples for the DFAs of concerned procedures. This reduces the inference of SPAs to a simultaneous inference of the DFAs for the involved procedures for which we can utilize various existing regular learning algorithms. The inferred models are easy to understand and allow for an intuitive display of the procedural system under learning that reveals its recursive structure. We implemented the algorithm within the LearnLib framework in order to provide a ready-to-use tool for practical application which is publicly available on GitHub for experimentation.

Keywords Active automata learning · Procedural systems · Context-free languages · Visibly pushdown languages

1 Introduction

Formal validation and verification methods such as model-based testing [10] and model-checking [6,14] are an integral part of today's software development process. As software grows in size and complexity, (automated) validation and verification of system properties not only helps finding errors during the development process but often is a requirement for final acceptance tests.

Crucial for these techniques to be applied properly is a formal model of the system (components) to verify. However, one often faces situations where formal representations are not available, either because creating and maintaining a correct model is tedious and error-prone or not possible at all if dealing with legacy systems or third-party components. Active Automata Learning (AAL) has shown to be a powerful means to attack these problems in many applications by allowing to infer behavioral models fully automatically on the basis of testing [24,31,35,41].

The fact that AAL as a testing-based technology is neither correct nor complete can nicely be compensated for via monitoring-based lifelong learning which became practical with the development of the TTT algorithm [29]. Essential for the success of AAL is the availability of powerful tools like [9,13,30,46] and the continuous development of more and more expressive frameworks capturing increasingly many system properties like input/output behavior [25], data [1,8,15,22,23,27,28,34], probability [16], additional computational structures like hierarchy/recursion [26,33] and parallelism.

This paper presents a compositional approach to active automata learning of *Systems of Procedural Automata* (SPAs), an extension of Deterministic Finite Automata (DFAs) to systems of DFAs that can mutually call each other according to the classical copy-rule semantics (cf. Fig. 2) used already in early programming languages like Algol 60 [38]. SPAs are of high practical relevance as they allow one to efficiently learn intuitive recursive models of recursive

✉ Bernhard Steffen
steffen@cs.tu-dortmund.de

Markus Frohme
markus.frohme@cs.tu-dortmund.de

[1] Chair of Programming Systems, Faculty of Computer Science, TU Dortmund, Dortmund, Germany

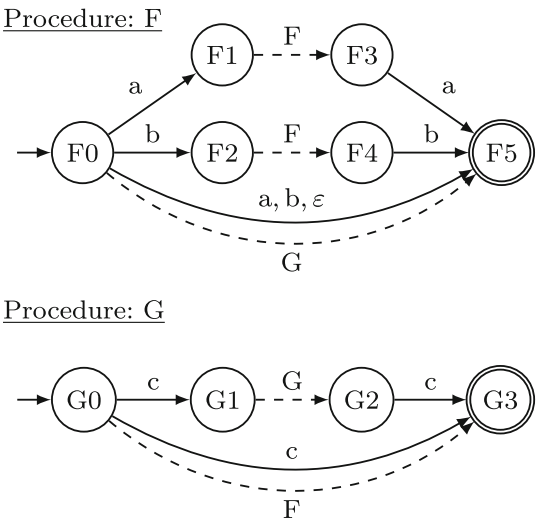

Fig. 1 An (NFA-based) SPA that captures the semantics of palindromes. Sink states and corresponding transitions of the NFAs are omitted for readability

programs after an easy instrumentation that makes calls and returns observable.

Key to our approach is the simultaneous inference of individual DFAs for each of the involved procedures in a modular fashion using well-known learning algorithms for deterministic finite automata [5,21,29,32,44]. Technically, our approach is based on a translation layer that bridges between the view of the entire system and the local view concerning the individual procedures: *Local queries* of procedural automata are expanded to *global queries* of the instrumented system under learning, and *global counterexample traces* for the global system are projected onto *local counterexample traces* for the concerned procedural automata. This translation layer introduces a negligible query overhead, as queries can be directly mapped between the two scopes and we show that counterexample projection can be implemented in a binary search-fashion.

Figure 1 illustrates three essential characteristics of SPAs:

- the intuitive structure: the operational semantics of SPAs follow the copy-rule semantics (cf. Fig. 2), i.e. upon encountering a procedural call the control is transferred to the respective procedural automaton from which it can only return at specific states. This is a universal concept that is independent of the automaton type of the procedures as it can be realized on a purely syntactical level, e.g. via graph transformation/rewriting [43].
  In this paper, we focus on a context-free language/acceptor-based perspective where successful runs of a system correspond to *accepted words* (wrt. the underlying language). Extending the SPA principle to other automaton types, such as transducers (e.g. Mealy machines), is on our future research agenda.
  The SPA in Fig. 1 is composed of non-deterministic finite automata (NFAs) to emphasize the trivial translation between SPAs and the production rules of context-free grammars (CFGs)—see Listing 1 for the corresponding representation in BNF.[1] In this paper, however, we focus on (equivalent) deterministic procedures (DFAs) which are more common in the active automata learning community. See Fig. 6 for the deterministic version of the SPA of Fig. 1. 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. 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. The SPA shown in Fig. 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.[2] 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.

```
F -> a | a F a | b | b F b | G | ε
G -> c | c G c | F
```

Listing 1 Production rules in BNF for the language of palindromes over the three characters `a`, `b` and `c`.

Similar to the classical learning of DFAs, our learning algorithm for SPAs terminates with a canonical SPA for the considered language and simply requires a so-called *membership oracle*, i.e. a "teacher" that is able to answer the required word problems—as long as one accepts that the so-called equivalence queries are typically approximated using membership queries in practice. Even better, also its (query) complexity remains unchanged as the effort is dominated by the learning of the individual procedural automata. As shown in Sect. 8, our approach yields significant performance improvements compared to existing learning algorithms for similar systems such as visibly pushdown automata (VPAs).

[1] In case a procedural automaton contains loops, the context-free grammar can be specified in extended BNF (EBNF) which allows regular expressions on the right-hand side of production rules.

[2] 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.

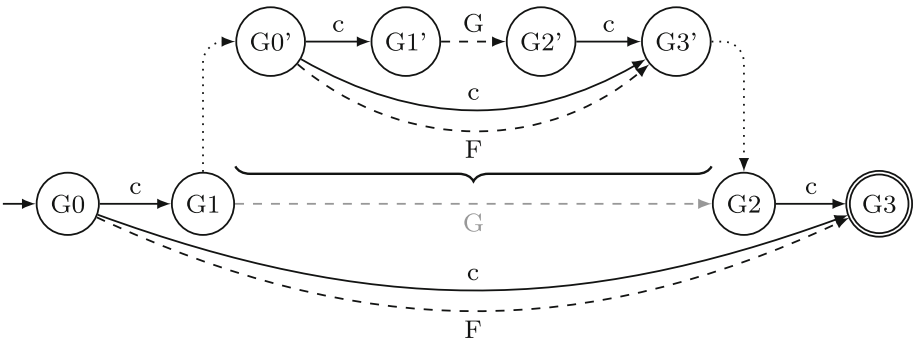

Fig. 2 The copy-rule semantics: for each procedural invocation the automaton of the called procedure is copied into the automaton of the calling procedure. This concrete example shows the first expansion step for the recursive invocation of G in procedure G. When labeling the dotted transitions with observable *call* and *return* symbols, the language of this (potentially infinite state) automaton coincides with the language of our instrumented system (cf. Listing 2). When interpreting them as direct (i.e. ε) transitions, the language coincides with the original context-free language

An implementation of the presented algorithm is publicly available at https://github.com/LearnLib/learnlib-spa and open to everyone for experimentation. Our implementation utilizes the LearnLib [30] library and therefore comes with direct support for practical application to real systems.

Outline

We continue in Sect. 2 with introducing the results of related fields of research and sketching preliminary terminology. Section 3 formalizes our concept of systems of procedural automata. In Sect. 4 we describe the essential concepts for the inference process of SPAs by formalizing the translation layer and the different phases of the learning process. Section 5 presents our approach to efficiently analyze and project global counterexamples for SPA hypotheses and Sect. 6 aggregates the previous concepts in a sketch of a learning algorithm. Sections 7 and 8 present a theoretical and empirical evaluation of the algorithm and Sect. 9 concludes the paper and provides some directions for future work.

2 Related work and preliminaries

The idea of SPAs was originally introduced under the name *Context-Free Process Systems* (CFPSs) in [11] for model checking and has since then been adapted to several similar formalisms such as *Recursive State Machines* (RSMs) in [2]. Calling them *Systems of Procedural Automata* here and focusing on deterministic automata is meant to better address the automata learning community. 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 [5]. 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.

The process of inferring a (regular) blackbox language is then given by discovering the equivalence classes of the Myhill-Nerode congruence [39] of the target language by means of partition refinement. This is usually done in an iterative loop consisting of the following two phases:

- exploration: the learning algorithm poses membership queries for exploring language characteristics and constructing a tentative hypothesis.
- verification: upon hypothesis stabilization, an equivalence query is posed that either indicates equivalence (thus terminating the learning process) or yields a counterexample (a word over the given input alphabet) which exposes a difference between the language of the tentative hypothesis and the unknown target language. This counterexample can then be used to refine the tentative hypothesis (by splitting a too coarse partition class) and start a new exploration phase.

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. [45] or [32, Chapter 8].

Regular languages are not powerful enough to capture the key characteristics of procedural systems which inherently support (potentially infinite) recursive calls between their sub-procedures. These semantics are, however, expressible with context-free languages. Angluin herself already reasoned about the inference of context-free languages [5], but her extensions required for answering, e.g., membership queries have—at least to the knowledge of the authors—prevented any practical application.

```
F' -> F a R | F a F' a R | F b R |
       F b F' b R | F G' R | F R
G' -> G c R | G c G' c R | G F' R
```

Listing 2 The palindrome system after our proposed instrumentation. Each procedure (F, G) is now treated as an observable call symbol indicating the start of a procedure and a separate return symbol (R) has been added to denote a procedure's termination. To maintain the hierarchiy of the original language, new non-terminals (X's) have been used.

For inferring context-free/procedural systems, we propose an instrumentation similar to the idea of parenthesis grammars [36]: Each invocation of a procedure can be observed by means of a *call symbol* which denotes the start of a specific procedure, as well as a *return symbol* which denotes its termination. An example of this instrumentation is given in Listing 2 which shows the instrumented system of palindromes of Listing 1.

This instrumentation can easily be integrated into software programs with imperative structure but also concepts such as aspect-oriented programming or proxying (in object-oriented programming) allow one to intercept method invocations and terminations and therefore grant the fine-grained control we require. For certain application domains—especially tag languages such as XML—these observable entry and exit points require no instrumentation at all. We exploit this property in [17] for inferring blackbox DTDs—a CFG-like language for describing the structure of XML documents.

The idea of assigning specific semantics to certain input symbols is conceptually related to *visibly pushdown languages* (VPLs) [3,4] proposed by Alur et al. Intrinsic to these languages is that the stack operations of the corresponding *visibly pushdown automaton* (VPA) are bound to the observation of certain symbols. The characterizations given by Alur et al. have been used by Kumar et al. [33] and Isberner [26, Chapter 6] to formulate learning algorithms for visibly pushdown languages, requiring only *classic* membership queries as well.

Alur et al. have shown that in general there exists no unique (up to isomorphism) minimal VPA for a visibly pushdown language without further restricting the automaton to a fixed set of modules. Therefore, they propose n-single entry visibly pushdown automata (n-SEVPAs) where the set of call symbols Σ_{call} is partitioned into n classes which are then "individually" represented in terms of n inter-connected structures. Such partitions support canonical representations, which in particular means that there exist canonical 1-SEVPA and $|\Sigma_{call}|$-SEVPA representations.

SPAs are conceptually close to $|\Sigma_{call}|$-SEVPAs and indeed, our proposed instrumentation transforms any context-free language into a visibly pushdown language, which allows us to compare the two approaches in Sect. 8. However, SPAs exhibit the following key advantages:

- The main difference between SPAs and n-SEVPAs concerns the treatment of the interrelation between substructures. Both in SPAs and n-SEVPAs, observing a call symbol guarantees to transition the system into a (for each call symbol) unique configuration. However, only for SPAs the same holds for observing the return symbol. While in general this allows n-SEVPAs to describe more complex languages, our instrumented systems do not require this complexity. Instead, a VPA learning algorithm has to compensate for this lack of certainty by posing more queries during the learning process. As Sect. 8 shows, this directly impacts the performance of the learning process, allowing our SPA learning algorithm to outperform the VPA approach by more than one order of magnitude in (symbol) query performance for small examples already.
- The generality of the SPA representation allows one to "implement" the semantics of an SPA via a variety of formalisms: SPAs can be realized via pushdown semantics (similar to VPAs), via in-place graph expansion (cf. Fig. 2), and directly via context-free grammars (e.g. using the CYK algorithm [20, Chapter 7]). They allow one to choose the best implementation for a specific situation, making them an implementation-agnostic meta model for context-free systems.
- The SPA structure is intuitive for everybody with programming knowledge and can directly be used also for μ-calculus-based model checking of context-free [11] and even pushdown processes [12], quite in contrast to the VPA representation and its "hard-coded" stack interpretation which is indeed quite cumbersome (cf. Sect. 8).
- As we will show, the compositional nature of SPAs allows us to learn the individual procedures with any learning algorithm for regular languages. Consequently, improvements in the field of regular language inference (e.g. the handling of redundancy in counterexamples in TTT [29]) seamlessly transfer to our procedural learning scenario.

The rest of this section summarizes formal definitions and notations we use throughout the paper.

Definition 1 (SPA alphabet) An *SPA alphabet* $\Sigma = \Sigma_{call} \uplus \Sigma_{int} \uplus \{r\}$ is the disjoint union of three finite sets, where Σ_{call} denotes the *call* alphabet, Σ_{int} 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* [3,4]. However, we choose a distinct name here in order to address the specifics of using only a single return symbol and to emphasize that *calling* a procedure does not necessarily involve any kind of stack operations. For our palindrome examples in Listings 1 and 2, the alphabet definition is given by $\Sigma = \{F, G\} \uplus \{a, b, c\} \uplus \{R\}$. We

write Σ^* to denote the set of all words over an alphabet Σ and we use $\cdot$ to denote the concatenation of symbols and words.

Furthermore, we distinguish between global and procedural interpretations of words and symbols, where we use $\widehat{\ }$ to denote the procedural context. We write $\widehat{\Sigma} = \widehat{\Sigma}_{call} \uplus \widehat{\Sigma}_{int} \uplus \{\widehat{r}\}$ and $\widehat{w} \in \widehat{\Sigma}^*$, accordingly. We add (remove) $\widehat{\ }$ to (from) individual words or symbols in order to change the context of a word or symbol to a procedural (global) one. We continue to use Σ as a shorthand notation for $\Sigma_{call} \uplus \Sigma_{int} \uplus \{r\}$ and $\widehat{\Sigma}$ as a shorthand notation for $\widehat{\Sigma}_{call} \uplus \widehat{\Sigma}_{int} \uplus \{\widehat{r}\}$.

In the following, we are especially interested in subwords: For $1 \leq i \leq j \leq |w|$, where $|w|$ denotes the length of a word 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). We write $w[i,]$ ($w[, j]$) to denote the suffix starting at position i (prefix up to and including position j). For any $i > j$, $w[i, j]$ denotes the empty word ε.

Due to our proposed instrumentation, we focus especially on *well-matched* instrumented words. 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 such that a *well-matched* nesting structure is obtained. Formally, we define the set of well-matched words by induction.

Definition 2 (Well-matched words) Let Σ be an SPA alphabet. We define the set of well-matched words $WM(\Sigma) \subset \Sigma^*$ as the smallest set satisfying the following properties:

- Every word of only internal symbols is well-matched, i.e. $\Sigma_{int}^* \subseteq WM(\Sigma)$.
- If $w \in WM(\Sigma)$ then for all $c \in \Sigma_{call}$ we have $c \cdot w \cdot r \in WM(\Sigma)$
- If $w_1, w_2 \in WM(\Sigma)$ then $w_1 \cdot w_2 \in WM(\Sigma)$.

We call well-matched words *rooted* if they start with a call symbol and end with a return symbol. In order to specify the scope of a procedural subsequence and to talk about the depth of nested procedural invocations, we further introduce the concept of a call-return balance.

Definition 3 (Call-return balance) Let Σ be an SPA alphabet. The call-return balance is a function $\beta\colon \Sigma^* \to \mathbb{Z}$, defined as

$$\beta(\varepsilon) = 0$$

$$\beta(u \cdot v) = \beta(v) + \begin{cases} 1 & \text{if } u \in \Sigma_{call} \\ 0 & \text{if } u \in \Sigma_{int} \\ -1 & \text{if } u = r \end{cases}$$

for all $u \in \Sigma, v \in \Sigma^*$.

For a well-matched word $w \in WM(\Sigma)$ we have that every prefix u satisfies $\beta(u) \geq 0$ and every suffix v satisfies $\beta(v) \leq 0$.

We further introduce a *find-return* function that allows us to extract the earliest unmatched return symbol from a (sub-) word, and an *instances set* that describes all call symbols and their respective indices in a word.

Definition 4 (Find-return function) Let Σ be an SPA alphabet and $w \in \Sigma^*$. We define the *find-return* function $\rho_w\colon \mathbb{N} \to \mathbb{N}$ as

$$\rho_w(x) = \min\{i \in \{x, \ldots, |w|\} \mid \beta(w[x, i]) < 0\}$$

Definition 5 (Instances set) Let Σ be an SPA alphabet and $w \in \Sigma^*$. We define the *instances set* $Inst_w \subseteq \Sigma_{call} \times \mathbb{N}$ as

$$Inst_w = \{(c, i) \mid w[i] = c \in \Sigma_{call}\}$$

3 Systems of procedural automata

In this section we present the base formalism of our approach: orchestrating regular systems to a system of procedural automata.

3.1 Orchestrating regular DFAs to procedural systems

We start with introducing *procedural automata* which form the core components of our systems of automata. Intuitively, they describe the possible actions of a single procedure and therefore an essential part of the global system behavior.

Definition 6 (Procedural automaton) Let Σ be an SPA alphabet and $c \in \Sigma_{call}$ denote a procedure. A procedural automaton for procedure c over Σ is a deterministic finite automaton $P^c = (Q^c, q_0^c, \delta^c, Q_F^c)$, where

- Q^c denotes the finite, non-empty set of states,
- $q_0^c \in Q^c$ denotes the initial state,
- $\delta^c\colon Q^c \times (\widehat{\Sigma}_{call} \cup \widehat{\Sigma}_{int}) \to Q^c$ denotes the transition function, and
- $Q_F^c \subseteq Q^c$ denotes the set of accepting states.

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

In essence, procedural automata resemble regular DFAs over the joined alphabet of call symbols and internal symbols and accept the language of right-hand sides of the production rules of a non-terminal in a (non-instrumented) context-free grammar. Internal symbols correspond to "terminal" actions, i.e. direct system actions, whereas call symbols correspond to (recursive) "calls" to other procedures. Please note that accepting states are used here to express that a procedure can terminate after a sequence of actions instead of using the (artificial) return symbol.

Definition 7 (System of procedural automata) Let Σ be an SPA alphabet with $\Sigma_{call} = \{c_1, \ldots, c_q\}$. A *system* of procedural automata S over Σ is given by the tuple of procedural automata $(P^{c_1}, \ldots, P^{c_q})$ such that for each call symbol there exists a corresponding procedural automaton. The *initial* procedure of S is denoted as $c_0 \in \Sigma_{call}$.

An example of such a system of procedural automata is given by the two DFAs in Fig. 6. We will continue to use S as a shorthand notation for $(P^{c_1}, \ldots, P^{c_q})$.

Intuitively, the parallels between SPAs and context-free grammars should be clear to everyone with a basic understanding of context-free languages. To formally define the *language* of an SPA, we use *structural operational semantics* (SOS, [42]), incorporating stack semantics. Using an SOS-based semantic definition allows us to abstract from implementation details (e.g. graph-transformations, grammar-based interpretation, etc.) and simplifies the proofs. We write

$$\frac{guard}{(s_1, \sigma_1) \xrightarrow{o} (s_2, \sigma_2)}$$

for some states s_1, s_2 and some control components σ_1, σ_2 to denote that this transformation (if applicable) *emits* an output symbol o. We generalize this notation to output *sequences* by writing

$$(s_1, \sigma_1) \xrightarrow{w}{}^{*} (s_2, \sigma_2)$$

to denote that there exists a sequence of individual (applicable) transformations starting in configuration (s_1, σ_1) and ending in configuration (s_2, σ_2), whose concatenation of output symbols yields w.

To define the semantics of SPAs by means of SOS rules, we first define a *stack* to model the control components of the SOS rules and then define the language of an SPA.

Definition 8 (Stack domain/configuration) Let Σ be an SPA alphabet. We define $\Gamma = \widehat{\Sigma}^* \uplus \{\bot\}$ as the *stack domain* with $\bot$ as the unique bottom-of-stack symbol. We use $\bullet$ to denote the stacking of elements of Γ where writing elements left-to-right displays the stack top-to-bottom and we write $ST(\Gamma)$ to denote the set of all possible *stack configurations*.

This allows us to define the semantics of an SPA in terms of its associated language.

Definition 9 (Language of an SPA) Let Σ be an SPA alphabet and S be an SPA over Σ. Using tuples from $\widehat{\Sigma}^* \times ST(\Gamma)$ to denote a system configuration, we define three kinds of SOS transformation rules:

1. call-rules:

$$\frac{\widehat{w} \in L(P^c)}{(\widehat{c} \cdot \widehat{v}, \sigma) \xrightarrow{c} (\widehat{w} \cdot \widehat{r}, \widehat{v} \bullet \sigma)}$$

for all $\widehat{c} \in \widehat{\Sigma}_{call}$, $\widehat{v} \in \widehat{\Sigma}^*$, $\widehat{w} \in (\widehat{\Sigma}_{call} \cup \widehat{\Sigma}_{int})^*$, $\sigma \in ST(\Gamma)$.

2. int-rules:

$$\frac{-}{(\widehat{i} \cdot \widehat{v}, \sigma) \xrightarrow{i} (\widehat{v}, \sigma)}$$

for all $\widehat{i} \in \widehat{\Sigma}_{int}$, $\widehat{v} \in \widehat{\Sigma}^*$, $\sigma \in ST(\Gamma)$.

3. ret-rules:

$$\frac{-}{(\widehat{r}, \widehat{v} \bullet \sigma) \xrightarrow{r} (\widehat{v}, \sigma)}$$

for all $\widehat{v} \in \widehat{\Sigma}^*$, $\sigma \in ST(\Gamma)$.

The language of an SPA is then defined as

$$L(S) = \{w \in WM(\Sigma) \mid (\widehat{c_0}, \bot) \xrightarrow{w}{}^{*} (\varepsilon, \bot)\}.$$

Please note that by choosing $(\widehat{c_0}, \bot)$ as the initial configuration, we ensure that all words of a (non-empty) SPA language are rooted because of the mandatory initial application of a call-rule to consume $\widehat{c_0}$. However, if for example $L(P^{c_0}) = \emptyset$, we have $L(S) = \emptyset$ as well.

To give an intuition for the operational semantics let us give an exemplary run for the SPA $S = (P^F, P^G)$ of Fig. 6, using F as the initial procedure: We start with the configuration $(\widehat{F}, \bot)$. Since $\widehat{a} \cdot \widehat{F} \cdot \widehat{a} \in L(P^F)$, we can apply a call-rule to perform the transition

$$(\widehat{F}, \bot) \xrightarrow{F} (\widehat{a} \cdot \widehat{F} \cdot \widehat{a} \cdot \widehat{R}, \varepsilon \bullet \bot).$$

Parsing the internal symbol $\widehat{a}$ via the corresponding int-rule, we perform

$$(\widehat{a} \cdot \widehat{F} \cdot \widehat{a} \cdot \widehat{R}, \varepsilon \bullet \bot) \xrightarrow{a} (\widehat{F} \cdot \widehat{a} \cdot \widehat{R}, \varepsilon \bullet \bot).$$

Since $\widehat{G} \in L(P^F)$, we can apply a call-rule to perform the transition

$$(\widehat{F} \cdot \widehat{a} \cdot \widehat{R}, \varepsilon \bullet \bot) \xrightarrow{F} (\widehat{G} \cdot \widehat{R}, \widehat{a} \cdot \widehat{R} \bullet \varepsilon \bullet \bot).$$

Since $\widehat{c} \in L(P^G)$, we can apply a call-rule to perform the transition

$$(\widehat{G} \cdot \widehat{R}, \widehat{a} \cdot \widehat{R} \bullet \varepsilon \bullet \bot) \xrightarrow{G} (\widehat{c} \cdot \widehat{R}, \widehat{R} \bullet \widehat{a} \cdot \widehat{R} \bullet \varepsilon \bullet \bot).$$

Parsing the internal symbol $\widehat{c}$ via the corresponding int-rule, we perform

$$(\widehat{c} \cdot \widehat{R}, \widehat{R} \bullet \widehat{a} \cdot \widehat{R} \bullet \varepsilon \bullet \bot) \xrightarrow{c} (\widehat{R}, \widehat{R} \bullet \widehat{a} \cdot \widehat{R} \bullet \varepsilon \bullet \bot).$$

Now we use two ret-rules to parse two consecutive return symbols

$$(\widehat{R}, \widehat{R} \bullet \widehat{a} \cdot \widehat{R} \bullet \varepsilon \bullet \bot) \xrightarrow{R} (\widehat{R}, \widehat{a} \cdot \widehat{R} \bullet \varepsilon \bullet \bot) \xrightarrow{R} (\widehat{a} \cdot \widehat{R}, \varepsilon \bullet \bot).$$

Parsing the internal symbol $\widehat{a}$ via the corresponding int-rule, we perform

$$(\widehat{a} \cdot \widehat{R}, \varepsilon \bullet \bot) \xrightarrow{a} (\widehat{R}, \varepsilon \bullet \bot).$$

Applying a ret-rule again, we get

$$(\widehat{R}, \varepsilon \bullet \bot) \xrightarrow{R} (\varepsilon, \bot).$$

Here, no more transformations are applicable and the process stops. Collapsing these individual steps, we have

$$(\widehat{F}, \bot) \xrightarrow{F \cdot a \cdot F \cdot G \cdot c \cdot R \cdot R \cdot a \cdot R} {}^{*}(\varepsilon, \bot)$$

hence $F \cdot a \cdot F \cdot G \cdot c \cdot R \cdot R \cdot a \cdot R \in L(S)$.

In the following, we will call a word $w \in \Sigma^*$ *admissible* in an SPA S, if there exist configurations (s_1, σ_1), $(s_2, \sigma_2) \in \Sigma^* \times ST(\Gamma)$ such that

$$(s_1, \sigma_1) \xrightarrow{w}{}^{*} (s_2, \sigma_2).$$

Please note, while the language of an SPA consists only of instrumented words, the non-instrumented language of the original context-free language (cf. Listing 2) can be easily obtained by post-processing each word of the SPA language and removing each of the instrumentation symbols $(\Sigma_{call} \cup \{r\})$, i.e. transforming $F \cdot a \cdot F \cdot G \cdot c \cdot R \cdot R \cdot a \cdot R$ to $a \cdot c \cdot a$.

4 Essentials of SPA inference

As Theorem 1 will show, an SPA is fully characterized by its procedural automata. Therefore, the key idea of our learning algorithm for inferring an SPA is to infer each of the procedures/procedural automata simultaneously by using an active learning algorithm for the individual (regular) procedural languages. Within the MAT framework, this requires that the local learning algorithms can explore the local procedures of a global system and that (global) counterexample information can be returned to the local learners.

Key to our approach is a translation layer that bridges between the global system view and the local view concerning the individual procedural automata: *Local queries* of procedural automata are expanded to *global queries* of the instrumented SUL, and *global counterexample traces* for the global system are projected onto *local counterexample traces* for the concerned procedural automata.

To be 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, 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).

For a procedure $c \in \Sigma_{call}$, we formalize the pairs of access and return sequences by means of a *context* $Cont_c$ and the set of successfully terminating runs by means of a set TS_c.

Definition 10 (Access, terminating, and return sequences) Let Σ be an SPA alphabet and S be an SPA over Σ. The context of a procedure $c \in \Sigma_{call}$, $Cont_c \subseteq \Sigma^* \times \Sigma^*$, and the set of terminating sequences $TS_c \subseteq \Sigma^*$ are defined as

$$\begin{aligned}
&(as, rs) \in Cont_c \wedge ts \in TS_c \Leftrightarrow \\
&\quad \exists w \in L(S) \colon \exists (c, i) \in Inst_w \colon \\
&\quad w = as \cdot ts \cdot rs \wedge \\
&\quad ts = w[i+1, \rho_w(i+1) - 1]
\end{aligned}$$

From the exemplary word $F \cdot a \cdot F \cdot G \cdot c \cdot R \cdot R \cdot a \cdot R$ of Sect. 3 one can extract the following access, terminating and returning sequences for procedure G:

- access sequence: $F \cdot a \cdot F \cdot G$
- terminating sequence: c
- return sequence: $R \cdot R \cdot a \cdot R$

Since an SPA can accept multiple words and a procedure can be called multiple times, there can also exist multiple terminating sequences and (matching) access and return sequence pairs. In the following, we do not depend on a specific instance of the three sequences as long as they hold the above properties. To refer to an arbitrary instance of an access, terminating and (matching) return sequence for a procedure p, we write $as[p]$, $ts[p]$, $rs[p]$ respectively. We detail in Sect. 6 how we obtain these sequences throughout the learning process—in the following, assume them as available.

We continue to explain in the next two subsections how our two translations are realized, where we begin with the simpler query expansion:

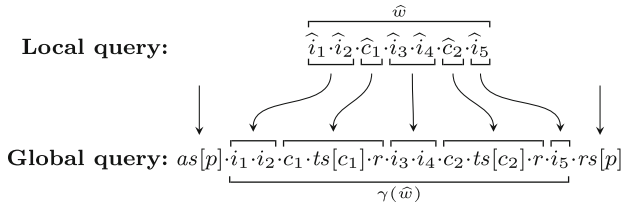

Fig. 3 The expansion of a local query of a procedural automaton p to a global query of the instrumented SUL

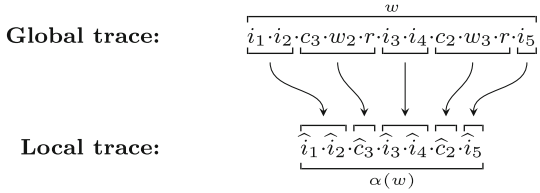

Fig. 4 The alpha projection of a global trace to a local trace. For our exemplary word $F \cdot a \cdot F \cdot G \cdot c \cdot R \cdot R \cdot a \cdot R$ of Sect. 3, projecting the outer-most run of F would yield $\alpha(a \cdot F \cdot G \cdot c \cdot R \cdot R \cdot a) = \widehat{a} \cdot \widehat{F} \cdot \widehat{a}$

Membership query expansion Membership query expansion proceeds by symbol-wise processing of the proposed (local) query. Internal symbols are left unchanged and each (procedrual) call symbol $\widehat{c} \in \widehat{\Sigma}_{call}$ is replaced with the concatenation of its global equivalent $c \in \Sigma_{call}$, the corresponding terminating sequence of c, and the return symbol. This expansion step is formalized in Definition 11.

Definition 11 (Gamma expansion) Let Σ be an SPA alphabet. The *gamma expansion* $\gamma : (\widehat{\Sigma}_{call} \cup \widehat{\Sigma}_{int})^* \to WM(\Sigma)$ is defined as

$$\gamma(\varepsilon) = \varepsilon$$
$$\gamma(\widehat{u} \cdot \widehat{v}) = \begin{cases} u \cdot \gamma(\widehat{v}) & \text{if } \widehat{u} \in \widehat{\Sigma}_{int} \\ u \cdot ts[u] \cdot r \cdot \gamma(\widehat{v}) & \text{if } \widehat{u} \in \widehat{\Sigma}_{call} \end{cases}$$

for all $\widehat{u} \in (\widehat{\Sigma}_{call} \cup \widehat{\Sigma}_{int})$, $\widehat{v} \in (\widehat{\Sigma}_{call} \cup \widehat{\Sigma}_{int})^*$.

In order to embed an expanded query into the correct global context, we further prepend the corresponding access sequence and append the corresponding return sequence to the translated query of the procedure in question. The complete expansion step is illustrated in Fig. 3.

Counterexample projection During counterexample analysis (cf. Sect. 4.3), one extracts from a global, instrumented trace of the SUL a sub-sequence that exposes wrong behavior in one of the procedural hypotheses. This sub-sequence, however, contains symbols of our instrumentation, which we need to transform to a local, procedural counterexample in order to be processable by the local learner. The corresponding projection step which essentially reverses query expansion is a bit more involved.

Again, we process the global trace symbol-wise and leave internal symbols unchanged. However, when we encounter an instrumented call symbol $c \in \Sigma_{call}$, we replace it with the corresponding procedural call symbol $\widehat{c} \in \widehat{\Sigma}_{call}$ and skip forward until we have reached the matching return symbol. This procedure is formalized in Definition 12.

Definition 12 (Alpha projection) Let Σ be an SPA alphabet. The *alpha projection* $\alpha : WM(\Sigma) \to (\widehat{\Sigma}_{call} \cup \widehat{\Sigma}_{int})^*$ is defined as

$$\alpha(\varepsilon) = \varepsilon$$
$$\alpha(u \cdot v) = \begin{cases} \widehat{u} \cdot \alpha(v) & \text{if } u \in \Sigma_{int} \\ \widehat{u} \cdot \alpha(v[\rho_v(1) + 1,]) & \text{if } u \in \Sigma_{call} \end{cases}$$

for all $u \in (\Sigma_{call} \cup \Sigma_{int})$, $v \in \Sigma^*$.

Note that since α only accepts well-matched words, the call to ρ will always be able to find a valid return index in v when u is a call symbol. Furthermore, u will never be a return symbol, because we always skip over any nested return symbols in case of $u \in \Sigma_{call}$. See Fig. 4 for an illustration of this projection step.

4.1 Localization theorem

With the concept of projection (cf. Definition 12) we establish in Theorem 1 a characteristic connection between the global language of an SPA and the local languages of its individual procedures, which will play an integral role in our inference process.

Theorem 1 (Localization theorem) *Let Σ be an SPA alphabet, S be an SPA over Σ and $w \in WM(\Sigma)$ be rooted in c_0.*

$$w \in L(S) \Leftrightarrow$$
$$\forall (c, i) \in Inst_w : \alpha(w[i+1, \rho_w(i+1) - 1]) \in L(P^c)$$

Proof This equivalence is based on the fact that for every emitted call symbol c of the SPA, there needs to exist a corresponding word $\widehat{v} \in L(P^c)$. One can verify this property for each call symbol by checking the membership of the projected, procedural trace in the language of the respective procedural automaton. For the full proof, see "Appendix" □

Theorem 1 guarantees that a word $w \in WM(\Sigma)$ which is rooted in the initial procedure belongs to the language of an SPA if and only if each (projected) procedural subsequence within w belongs to the language of its respective procedural automaton. It is important to note that this equivalence establishes a notion of modularity between an SPA and

its procedural automata: Procedural automata contribute to the semantics of an SPA only by their respective procedural membership properties—no further requirements (e.g. about their internal structure) are needed. In particular, this property enables us to use arbitrary (MAT-based) regular learners for inferring the procedural automata and, consequently, for inferring an SPA.

Furthermore, we can show that our SPA approach can integrate to the MAT framework as well, by demonstrating how to realize the *exploration* and *verification* phase of the MAT framework.

4.2 Exploration phase

In Corollary 1 we formalize that each local membership query $\widehat{w} \in (\widehat{\Sigma}_{call} \cup \widehat{\Sigma}_{int})^*$ can be answered by querying the global SPA with its expanded version.

Corollary 1 (Membership query expansion) *Let Σ be an SPA alphabet and S be an SPA over Σ.*

$$\widehat{w} \in L(P^c) \Leftrightarrow as \cdot \gamma(\widehat{w}) \cdot rs \in L(S)$$

for all $c \in \Sigma_{call}$, $(as, rs) \in Cont_c$.

Proof This equivalence is based on the fact that pairs of access sequences and matching return sequences for a procedure c provide an admissible context for arbitrary $\widehat{w} \in L(P^c)$. One can then show by induction that for all $\widehat{w} \in L(P^c)$, $(\widehat{w} \cdot \widehat{r}, \sigma) \xrightarrow{\gamma(\widehat{w})}{}^* (\widehat{r}, \sigma)$ holds (for some $\sigma \in ST(\Gamma)$). For the full proof, see "Appendix" □

By providing the local learners with individual membership oracles that perform these translations automatically, the exploration phase of the global SPA hypothesis is directly driven by the exploration phases of the individual local learners. In order to construct and explore an SPA hypothesis $S_{\mathcal{H}}$, we use the procedural learners to construct and explore hypotheses of the individual procedures. What remains to be shown is how the information of global counterexamples can be used to refine local procedures.

4.3 Verification phase

Counterexamples are input sequences that expose differences between the conjectured SPA hypothesis and the SUL as they reveal diverging behavior with regards to the membership question. For acceptor-based systems there exist two kinds of counterexamples: *positive* and *negative* counterexamples. Positive counterexamples are words which are accepted by the SUL but (incorrectly) rejected by the current hypothesis, whereas negative counterexamples are rejected by the SUL but (incorrectly) accepted by the hypothesis.

In the following, we will abstract from the concrete kind of counterexample and only consider an *accepting* system $S_A = (P_A^{c_1}, \cdots, P_A^{c_q})$ and a *rejecting* system $S_R = (P_R^{c_1}, \cdots, P_R^{c_q})$. For positive counterexamples we map the SUL to S_A and the current hypothesis to S_R and for negative counterexamples we do the converse. This shows that, conceptually, the counterexample analysis process for SPAs is symmetrical for both kinds of counterexamples and that the kind only determines the mapping of S_R and S_A. We will, however, see that this symmetry does not hold when considering the query complexity.

Given a counterexample ce (and therefore S_A and S_R), Theorem 1 states that

$$\forall (c, i) \in Inst_{ce} \colon \alpha(ce[i+1, \rho_{ce}(i+1) - 1]) \in L(P_A^c)$$

and

$$\exists (c, i) \in Inst_{ce} \colon \alpha(ce[i+1, \rho_{ce}(i+1) - 1]) \notin L(P_R^c).$$

This allows us to split the counterexample analysis process into two phases:

1. In the *global phase*, we first analyze the global counterexample to pinpoint an individual procedure of S_R that behaves differently from its respective counterpart in S_A.
2. In the *local phase*, we use the corresponding projected sub-sequence of the global counterexample to refine the previously identified procedure.

Due to our concept of projection and expansion, the refinement during the local phase is essentially identical to the refinement phase of regular automata learning. Consequently, we can delegate this process completely to the local learning algorithms and focus in the following only on the first phase. The goal of our global counterexample analysis is then given by identifying a (not necessarily unique) procedural automaton which does not accept its corresponding projected trace and causes a mismatching behavior.

Once we have identified the misbehaving procedural automaton, we can use the projected trace to construct a local counterexample: In the positive case (i.e. the hypothesis is mapped to S_R and hence should accept the local trace) we construct a positive local counterexample and in the negative case (i.e. the SUL is mapped to S_R and hence the hypothesis should also reject the local trace) we construct a negative local counterexample.

As mentioned, the projected counterexample is completely agnostic to the internal structures of the procedural hypotheses and only relies on their respective membership properties. Thus, following Theorem 1, we can use arbitrary (MAT-based) regular learning algorithms for refining the procedural hypotheses.

Please note, however, that Theorem 1 only guarantees the existence of a procedure that needs refinement. Efficiently identifying such a procedure is a different matter that we address in the next section.

5 Efficient counterexample analysis

In order to efficiently analyze global counterexamples and identify a procedure of S_R that rejects its projected trace, we propose a binary search-based approach similar to the one of Rivest & Schapire [44] for the regular case. This process consists of transforming prefixes of the counterexample to sequences that are guaranteed to be admissible in S_R and analyzing when this transformation changes the acceptance of the transformed counterexample trace.

Our approach to make binary search applicable in our scenario depends on S_R to accept the current representatives of terminating sequences $ts[p]$ that are used by the gamma expansion (cf. Definition 11). This is naturally given for the SUL, as we will extract terminating sequences only from accepted runs of the SUL (cf. Sect. 6), but has to be explicitly enforced for the hypothesis. We therefore introduce the following notion:

Definition 13 (ts-conformance) Let Σ be an SPA alphabet and S be an SPA over Σ. We call S *ts-conform* with respect to the current terminating sequences if and only if $\forall c \in \Sigma_{call} \colon \exists \widehat{w} \in L(P^c) \colon$

$$(\widehat{w} \cdot \widehat{r}, \sigma) \xrightarrow{ts[c]}{}^{*} (\widehat{r}, \sigma)$$

for some $\sigma \in ST(\Gamma)$.

For testing and validating the ts-conformance of an SPA, we re-use the result of Theorem 1.

Lemma 1 *Let Σ be an SPA alphabet, S be an SPA over Σ and $ts_c = c \cdot ts[c] \cdot r$ denote the embedded terminating sequence for each $c \in \Sigma_{call}$.*

$$\begin{aligned} &S \text{ is ts-conform} \Leftrightarrow \\ &\forall p \in \Sigma_{call} \colon \forall (c, i) \in Inst_{ts_p} \colon \\ &\alpha(ts_p[i+1, \rho_{ts_p}(i+1) - 1]) \in L(P^c) \end{aligned}$$

Proof This is a direct consequence of Theorem 1 if we consider for each $p \in \Sigma_{call}$ an SPA S_p (based on S) which has p as its initial procedure. □

Enforcing ts-conformance of an SPA hypothesis $S_{\mathcal{H}}$ can be done straightforwardly as follows:

- Check for each embedded terminating sequence ts_p whether its nested, projected invocations are accepted by the respective procedures of $S_{\mathcal{H}}$. This does not require any membership queries since it can be checked on the procedural hypothesis automata.
- If there exists a (nested) invocation that is not accepted, use the corresponding sequence as a local counterexample for refining the corresponding procedural hypothesis. This can be regarded as a "refinement for free", as it does not require the detection and treatment of a global counterexample.

Please note that this conformance check has to be re-initiated after each refinement, as refining the procedural hypotheses may introduce changes that affect the acceptance of a terminating sequence.

Given a ts-conform system S_R the following transformation allows one to perform the intended binary search:

Definition 14 (Alpha-gamma transformation) Let Σ be an SPA alphabet and $w \in WM(\Sigma)$ be a well-matched word. Then we define $[\![\cdot]\!] \colon WM(\Sigma) \to WM(\Sigma)$ by

$$[\![w]\!] = \gamma(\alpha(w))$$

$[\![\cdot]\!]$ can be generalized to prefixes of rooted words

$$w_c = c_{i_1} \cdot w_1 \cdot \ldots \cdot c_{i_m} \cdot w_m \in (\Sigma_{call} \cdot WM(\Sigma))^*$$

to obtain a transformation

$$[\![\cdot]\!]^* \colon (\Sigma_{call} \cdot WM(\Sigma))^* \to (\Sigma_{call} \cdot WM(\Sigma))^*$$

defined via piecewise application of $[\![\cdot]\!]$ as follows:

$$\begin{aligned} [\![w_c]\!]^* &= [\![c_{i_1} \cdot w_1 \cdot \ldots \cdot c_{i_m} \cdot w_m]\!]^* \\ &= c_{i_1} \cdot [\![w_1]\!] \cdot \ldots \cdot c_{i_m} \cdot [\![w_m]\!] \end{aligned}$$

The following monotonicity property of $[\![\cdot]\!]^*$ is key to our binary search-based counterexample analysis. For technical reasons, we will, without loss of generality, only consider counterexamples with more than one procedural invocation. Please note that if a counterexample only contains the single invocation of the main procedure (i.e. the error occurs in the main procedure) there is no need for a global analysis process since the violating procedure is clear.

Theorem 2 (Acceptance monotonicity of $[\![\cdot]\!]^*$) *Let Σ be an SPA alphabet, S be a ts-conform SPA over Σ, $w \in WM(\Sigma)$ be rooted and r_h, r_k be indices of return symbols of w with $r_h < r_k$. Then we have*

$$\begin{aligned} &[\![w[, r_h]]\!]^* \cdot w[r_h + 1,] \in L(S) \Rightarrow \\ &\quad [\![w[, r_k]]\!]^* \cdot w[r_k + 1,] \in L(S) \end{aligned}$$

Proof This implication is based on the fact that for all admissible words $v \in WM(\Sigma)$ or $v \in (\Sigma_{call} \cdot WM(\Sigma))^*$, $[\![v]\!]^*$ is also admissible in a ts-conform SPA. Furthermore, the admissibility of a word is decided on call-rules, since they are the only rules which are guarded by the procedural membership questions. Hence, when the call symbols in $w[r_h+1,]$ do not cause a word to be rejected, then the call symbols of its suffix $w[r_k+1,]$ won't neither. For the full proof, see "Appendix" □

The acceptance monotonicity of $[\![\cdot]\!]^*$ allows us to adopt the Rivest & Schapire-style counterexample analysis of the regular case [44] to the procedural level: There exist two extreme points

$$[\![\varepsilon]\!]^* \cdot ce \notin L(S_R)$$

(the unprocessed counterexample) and

$$[\![ce]\!]^* \cdot \varepsilon \in L(S_R)$$

(the terminating sequence of the main procedure) so the acceptance has to flip for some decomposition in between.

For a return index r_i of ce, we check whether or not

$$[\![ce[,r_i]]\!]^* \cdot ce[r_i+1,] \in L(S_R).$$

If the answer is "yes", it suffices to search for lower return indices than r_i, because by Theorem 2, we already know the answers for all higher return indices. Dually, if the answer is "no", we continue our search for higher return indices than r_i, because by contraposition of Theorem 2, we already know the answers for all lower return indices.

This observation allows us to formulate a binary search-style analysis (cf. Algorithm 1) which determines the lowest return index r_* such that

$$[\![ce[,r_*]]\!]^* \cdot ce[r_*+1,] \in L(S_R).$$

Its matching call symbol c_* (whose index i_* can be determined by $\rho_{ce}(i_*+1) = r_*$) now identifies a procedure that does not accept its corresponding projected input sequence $\alpha(ce[i_*+1, r_*-1])$ and must therefore be refined. Please note that investigating a specific return index r_i only requires to query S_R once because the construction of $[\![ce[,r_i]]\!]^* \cdot ce[r_i+1,]$ can be done by in-memory transformations on ce.

To give a better intuition of this decomposition process, the first steps of the two cases (left-continuation and right-continuation of the binary search) are visualized in Fig. 5.

While the process of analyzing a global counterexample is symmetrical for the negative and positive case, it is worth noting that the two cases differ regarding their impact on the query complexity: For positive counterexamples we map the current SPA hypothesis to S_R. Here, determining c_* by querying S_R does not induce any membership queries at all, since the queries can be answered via the current SPA hypothesis. This means positive counterexamples can be analyzed at *zero* (query) cost. Only the analysis process of negative counterexamples requires to pose membership queries on the SUL, which introduces a corresponding logarithmic factor (cf. Theorem 3).

Summarizing, an inequivalent procedure and its corresponding local counterexample can be determined in the following steps:

1. Depending on whether we receive a positive or a negative (global) counterexample, we select either the current hypothesis SPA or the SUL as the rejecting system S_R.
2. Using S_R and our alpha-gamma transformation (cf. Definition 14), we determine a single procedural hypothesis that behaves differently to its counterpart of S_A.
3. Using the alpha projection (cf. Definition 12), we construct from the global counterexample a procedural counterexample that exposes the previously detected discrepancy on a procedural level.

We sketch these steps in a function called AnalyzeCounterexample shown in Algorithm 1. The function takes a global counterexample $ce \in WM(\Sigma)$ and returns the rejecting procedure c_* including the respective local counterexample $\alpha(ce[i_*+1, \rho_{ce}(i_*+1)-1])$. The main property of this function for our learning algorithm is stated in Theorem 3.

Algorithm 1 Analysis of a global counterexample

Input: A counterexample $ce \in WM(\Sigma)$ rejected by S_R
Output: A tuple containing a procedure c_* of S_R and its rejected, procedural trace

```
1: function AnalyzeCounterexample(ce)
2:     ensureTSconformance
3:     low ← 1, high ← |Inst_ce|, res ← |Inst_ce|
4:     while high − low ≥ 0 do
5:         mid ← low + ⌊(high − low)/2⌋
6:         if [[ce[, r_mid]]]* · ce[r_mid + 1,] ∈ L(S_R) then
7:             high ← mid − 1, res ← mid
8:         else
9:             low ← mid + 1
10:        end if
11:    end while
12:    i_* ← min{i ∈ ℕ | ρ_ce(i + 1) = r_res}
13:    return ⟨ce[i_*], α(ce[i_* + 1, r_res − 1])⟩
14: end function
```

Theorem 3 (Query complexity of counterexample analysis) *Let $ce \in WM(\Sigma)$ be a counterexample of length m. Then the query complexity of Algorithm 1 is $\mathcal{O}(\log_2 m)$.*

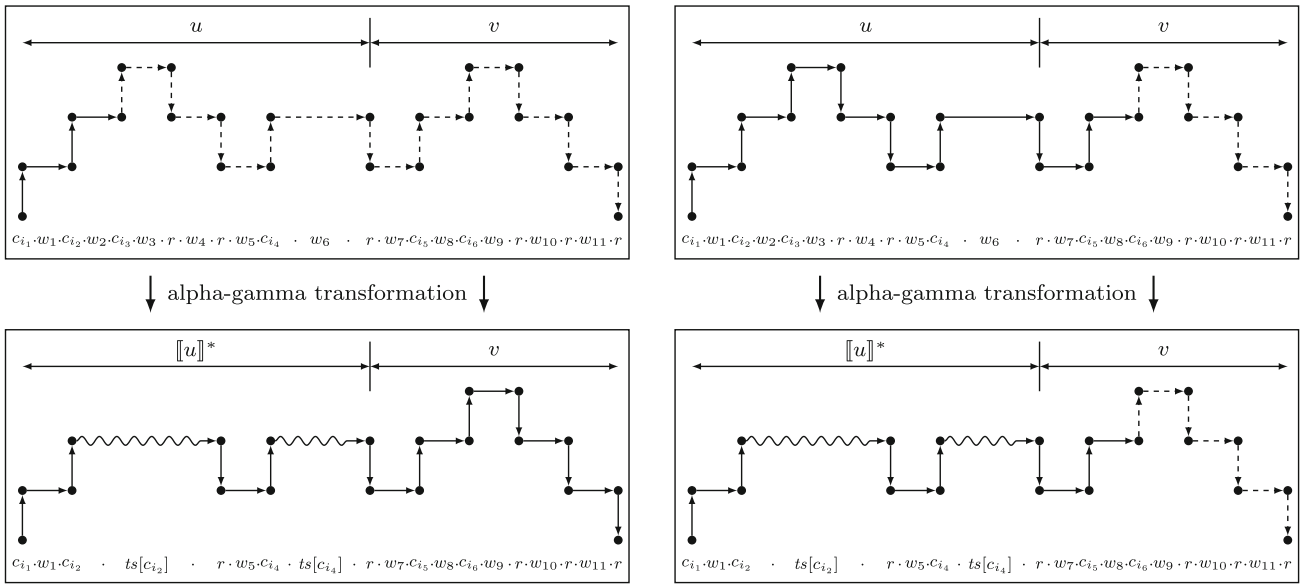

Fig. 5 The two possible scenarios during the analysis of counterexamples: the top two images each show a run of a counterexample trace in S_R, where dashed lines indicate that an illegal procedural invocation has occurred that irrecoverably causes S_R to reject the trace. On the left-hand side the error occurs in u. Here our (extended) alpha-gamma transformation replaces the violating procedural invocation with an admissible prefix, which causes the transformed trace to be accepted. This indicates that further analysis (i.e. binary search) should continue with splitting u. On the right-hand side the error occurs in v. Here the error prevails even after our (extended) alpha-gamma transformation, which indicates that further analysis (i.e. binary search) should continue with splitting v. Note that replacing nested calls via the alpha-gamma transformation may result in a more complex nesting structure than before, depending on the terminating sequence. In the above figure—for simplicity reasons—terminating sequences only consist of internal symbols

Proof This is a direct consequence of the binary search strategy and the fact that for a given return index r_{mid} the query $[\![ce[, r_{mid}]]\!]^* \cdot ce[r_{mid}+1,]$ can be constructed without any further membership queries. □

6 A sketch of the algorithm

In this section we aggregate the concepts from the previous sections and sketch an active learning algorithm for SPAs. As stated previously, we exploit that SPAs are characterized by their procedural automata which we can infer in a modular fashion by

- answering local membership queries via global membership queries to the SPA (cf. Sect. 4) and
- constructing local counterexamples for the procedures from global SPA counterexamples (cf. Sect. 5).

Query expansion hinges on the availability of corresponding access sequences, terminating sequences, and return sequences. Thus, one of the main tasks of the learning algorithm of an SPA is obtaining and managing these sequences throughout the learning process.

Positive counterexamples (i.e. words that are rejected by the current hypothesis but are accepted by the SUL) play a special role throughout the learning process because they are witnesses for successful runs of the SUL. In particular, since we are observing well-matched words, for every procedural invocation (i.e. call symbol) in a positive counterexample, we can directly extract:

- a corresponding access sequence (everything up until and including the call symbol),
- a terminating sequence for the procedure (everything in between the call symbol and the matching return symbol), and
- a return sequence for the procedure (the matching return symbol and everything after).

The following subsections display how we initialize our procedural learning algorithms and how we use positive counterexamples during hypothesis refinements to manage access sequences, terminating sequences and return sequences.

6.1 Initialization

We initialize our SPA learner by setting up regular learning algorithms (e.g. TTT [29]) for the procedures and configuring them accordingly (individual membership oracles for automated query translation, etc.). However, during the initial-

ization, the local learners cannot explore any procedures due to the lack of the required sequences mentioned above. As a consequence, it is also not possible to construct an initial SPA hypothesis (e.g. by means of applying a call-rule to the initial configuration of Definition 9). Instead, an initial (empty) hypothesis is constructed, that simply rejects all input words. This guarantees that the first counterexample our SPA learning algorithm receives will always be positive, which provides us with access sequences, terminating sequences and return sequences of at least the main procedure and ensures progress. If no such counterexample exists, i.e. the SUL describes the empty language ($L(SUL) = \emptyset \subseteq WM(\Sigma)$), the initial hypothesis already coincides with the SUL and the learning process is finished at this point.

6.2 Refinement

The core of the learning algorithm is the refinement step which for a given counterexample triggers a refinement of the hypothesis, or specifically in our case, a refinement of (at least) one procedure. Given the above initialization, it may however not be possible to address certain procedures due to the lack of initialization.

As we pointed out earlier, positive counterexamples hold a special role as they grant access to the required sequences for activating local learners and therefore constructing hypotheses of procedures. Generally, we cannot expect a single counterexample to contain all procedural call symbols at once and thereby giving us access to the information required for activating all local learners and reasoning about procedural invocations. Even after the first (global) counterexample there may be procedures for which no access, terminating or return sequences have been observed yet.

We tackle this issue by introducing the concept of *incremental alphabet extension*. In our context this means that we successively add call symbols to our learning alphabet only after we have witnessed them in a positive counterexample. This does not cause any problem because the learning process is monotonic wrt. alphabet extension.

At the start of the learning process, we initialize the currently active learning alphabet $\widehat{\Sigma}_{act}$ with $\widehat{\Sigma}_{int}$ so that it contains no call symbols. No local learning algorithms—not even for the start procedure—have been activated and thus the corresponding initial hypothesis specifies the empty language. As mentioned above, this guarantees that the first counterexample is positive (corresponds to a successful run) and therefore provides us with the three kinds of sequences for at least the start procedure. In general, gaining access to the three sequences of a procedure allows us

- in case of access and return sequences: to activate the corresponding local learning algorithm because its queries can now be embedded in a global context using access sequences and return sequences, and
- in case of terminating sequences: to invoke the corresponding procedure in other contexts because local queries can be properly expanded to admissible input sequence in the global system.

Essentially, we delay exploration of a procedure until we obtained the knowledge about its access sequence and return sequence, and restrict exploring a procedure wrt. the currently active alphabet which consists only of internal alphabet symbols and procedural invocations (i.e. call symbols) for which a terminating sequence has already been found. This guarantees that invocations of a procedure are reflected in the tentative SPA hypothesis only after they can be correctly embedded in the global context.

By rejecting words that contain uninitialized procedures, counterexamples that introduce previously unobserved call symbols will always be positive, allowing us to extract the three kinds of sequences and progress the SPA inference process. Thus the active alphabet successively grows towards the complete input alphabet $\widehat{\Sigma}_{call} \cup \widehat{\Sigma}_{int}$. Algorithm 2 describes the corresponding refinement process in more detail:

Lines 2 to 15 cover the aforementioned special handling of positive counterexamples. If we receive our first counterexample (cf. line 3), we extract c_0 from the counterexample. Recall that due to the nature of our instrumentation, all accepted words of the SUL are rooted in c_0 and thus the initial procedure can be directly determined from the first symbol of any accepted word of the SUL. We continue to scan the counterexample for previously unobserved call symbols (cf. line 6), extract the access sequences, terminating sequences and return sequences from the counterexample trace (cf. lines 7–9), and update the set of currently active alphabet symbols (cf. line 10). The variables as, ts, rs and $\widehat{\Sigma}_{act}$ are stored in a global scope and thus are shared across multiple invocations of the refinement procedure.

Discovering a (new) terminating sequence for a procedure p allows us to invoke p in the context of other procedures because local queries containing $\widehat{p}$ can now be correctly expanded to global queries using our gamma expansion (cf. Definition 11). Therefore, in line 13 we extend the alphabets of the procedural learners to match the set of currently active symbols $\widehat{\Sigma}_{act}$. For already activated procedural learners this includes posing new (local) queries in order to determine the successors of transitions for the just-added symbols.

From lines 16 to 19 we analyze the given counterexample with regards to the current hypothesis SPA $S_{\mathcal{H}}$. From every global counterexample trace ce we can extract a procedure c and a (projected) sub-sequence $localCe$ such that $localCe$ exposes an error in P^c (see Sect. 5). We then use the projected, local counterexample to delegate the actual refinement step of the identified local hypothesis to the respective

Algorithm 2 Main refinement loop of the SPA learner

Input: A counterexample $ce \in WM(\Sigma)$ and a boolean value $answer$ indicating whether ce is a positive or negative counterexample

```
1: function RefineHypothesis(ce, answer)
2:   if answer = true then
3:     if Σ̂_act = Σ̂_int then
4:       c_0 ← ce[1]                                  ▷ used for determining the initial procedure
5:     end if
6:     for all p ∈ DetectNewProcedures(ce, Σ_act) do   ▷ scan counterexample for new procedures
7:       as[p] ← ExtractAccessSequence(ce, p)
8:       ts[p] ← ExtractTerminatingSequence(ce, p)
9:       rs[p] ← ExtractReturnSequence(ce, p)
10:      Σ̂_act ← Σ̂_act ∪ {p̂}
11:    end for
12:    for all p ∈ Σ_call do
13:      ExtendProceduralAlphabets(p̂, Σ̂_act)          ▷ extend alphabets of procedural learners
                                                       with new terminating procedures and
                                                       complete the respective hypotheses
14:    end for
15:  end if
16:  while ce is a counterexample for S_H do
17:    ⟨c, localCe⟩ ← AnalyzeCounterexample(ce)
18:    RefineProcedure(P^c, localCe, answer)          ▷ delegate to local learner of P^c
19:  end while
20: end function
```

regular learning algorithm of the procedure (cf. line 18). Note that if the procedural learner of the determined procedure is not yet initialized, we initialize it first (to construct an initial hypothesis) and then pass the projected counterexample to the learner.

7 Correctness and complexity

A canonical SPA is given by the tuple $S = (P^{c_1}, \ldots, P^{c_q})$ such that each P^{c_i} is a canonical automaton for the corresponding procedure $c_i \in \Sigma_{call}$. The size of an SPA is the sum of the individual sizes of the procedures, i.e. the number of their states. We have

$$|S| = \sum_{i=1}^{q} |P^{c_i}| = \sum_{i=1}^{q} n_i = n.$$

Similar to the original work by Angluin [5], the following discussion assumes that so-called *equivalence queries* are available to indicate discrepancies between inferred hypothesis models and the considered SUL. In practice, equivalence queries are typically approximated using membership queries which themselves are realized via testing. The discussion of this issue, which is typically based on application specific heuristics (e.g. context-free model checking [11]), is beyond the scope of this paper.

Our following correctness and complexity considerations are based on the assumption that the individual procedural automata are learned using one of the well-known algorithms for regular inference which incrementally construct canonical (i.e. minimal unique, up to isomorphism) hypotheses requiring at most n_i (for procedure $c_i \in \Sigma_{call}$) equivalence queries. Under these assumptions one can show:

Theorem 4 (Correctness and termination) *Having access to a MAT teacher for an instrumented context-free language L, our learning algorithm determines a canonical SPA S for L requiring at most $n + 1$ equivalence queries.*

Proof The proof follows a three-step pattern:

- Invariance: The number of states of the hypothesis of procedure P^{c_i} never exceeds n_i, a central invariant of all learners for regular languages. Hence, the size of the SPA will never exceed n.
- Progress: Each global counterexample either activates a local learner, which then adds the first "true" state to its procedural hypothesis, or identifies an error in one of the existing tentative hypotheses. Using the presented counterexample analysis, one extracts from the global counterexample a concerned procedure c_i and the projected local counterexample that allows the local learner to refine the corresponding local hypothesis automaton for P^{c_i}. This adds at least one state to the procedural hypothesis of P^{c_i} and thereby properly increases the number of states of the hypothesis SPA. This refinement works by expanding the required local membership queries to SPA queries (cf. Fig. 3), and by interpreting the SPA responses to the expanded query as answers to the local query.
- Termination: After at most n_i local counterexamples the tentative hypothesis of c_i is equivalent to the target procedure. Hence, at most n_i global counterexamples can

identify an error in the hypothesis of c_i, which overall only allows for at most n global counterexamples. This is a direct consequence of the above notion of invariance and progress. The final (additional) equivalence query is required to confirm the equivalence of the SPA hypothesis and the SUL, and terminates the inference process.

□

We can further show that the query complexity of our algorithm, i.e. the number of posed membership queries, is mainly determined by the choice of the regular learning algorithms. Our concepts of orchestration (to systems of procedural automata) and query translation do not impact the asymptotic query complexity. For inferring an isolated procedure P^{c_i}, state-of-the-art learning algorithms (such as TTT [29]), require $\mathcal{O}(kn_i^2 + n_i \log_2 m)$ queries, where k denotes the size of the input alphabet (in our case $|\Sigma_{call}|+|\Sigma_{int}|$) and m denotes the length of the longest (local) counterexample. The query complexity is usually split into two parts: hypothesis construction $C_{c_i} = kn_i^2$ and counterexample analysis $n_i \log_2 m$. This results in the following query complexity.

Theorem 5 (Query complexity) *Let C_{c_i} denote the complexity for hypothesis construction of the local learner for procedure $c_i \in \Sigma_{call}$ and m denote the length of the longest (global) counterexample. Learning a canonical SPA S has a query complexity of $\mathcal{O}\left(\left(\sum_{i=1}^{q} C_{c_i}\right) + n \log_2 m\right)$.*

Proof This follows from the compositional nature of our algorithm and the following notion of progress throughout the learning process:

- Each global counterexample imposes at least one counterexample for the hypothesis of at least one procedural automaton and can be analyzed in binary search fashion (cf. Theorem 3).
- Each localized counterexample triggers the refinement of a procedural automaton and requires the amount of queries specific to the chosen local learning algorithm. The number of queries for the local counterexample analyses do not asymptotically impact the overall analysis complexity as

$$\begin{aligned}\sum_{i=1}^{q} \left(n_i \log_2 m\right) &= \left(\sum_{i=1}^{q} n_i\right) \log_2 m \\ &= n \log_2 m \\ &\in \mathcal{O}(n \log_2 m)\end{aligned}$$

- After all states of all procedures have been identified, the algorithm terminates with the correct hypothesis (cf. Theorem 4).

□

A direct comparison to existing learning algorithms for, e.g. visibly pushdown languages is hard, due to the different structure of the inferred models. However, first experiments (cf. Sect. 8) show that already for small systems our approach performs significantly better.

8 A comparison to visibly pushdown automata

To our knowledge, related work only considers VPAs. In order to elaborate on the qualitative and quantitative differences between our SPA approach and existing learning setups for visibly pushdown languages, we showcase the system of palindromes (cf. Fig. 1) as a visibly pushdown automaton. As stated before, the instrumented SUL yields a visibly pushdown language where every observable invocation can be interpreted as a call symbol and every observable termination can be interpreted as a return symbol. Thus, the instrumented system can also be inferred in form of a visibly pushdown automaton. For this showcase, we inferred the system of (instrumented) palindromes using learning algorithms currently present in LearnLib [30] which infer 1-SEVPAs.

The inferred models are shown in Figs. 6 and 7. Table 1 shows performance measurements of our SPA approach using different regular learning algorithms and the two VPA learners present in LearnLib. We approximated equivalence queries by generating 10000 random, well-matched, rooted test-words to allow for some variance and ultimately sampled from a manually constructed set of characteristic words to ensure that each algorithm terminated with the correct hypothesis model. As mentioned before, the realization of equivalence queries is an issue on its own, which is beyond of the scope of this paper.

Although both models capture in essence the same information, there is a big difference in their comprehensibility: Whereas the SPA model (Fig. 6) very intuitively reflects the underlying grammar and thus the structure of the system, the VPA model (Fig. 7) is quite hard to understand. For successfully following an accepting run of the automaton, one has to manually keep track of the current stack contents which have been pushed (popped) by previous call (return) symbols. In particular, it is hardly possible to reveal typical structural properties.

Regarding performance there is another interesting observation. While the 1-SEVPA representation is more compact in the sense that it can represent the system with fewer states/locations, it requires significantly more queries and even an order of magnitude more symbols to infer the model. We reckon this is due to the global execution semantics of SEVPAs: Essentially, every state can be the successor of a return transition and for every return tran-

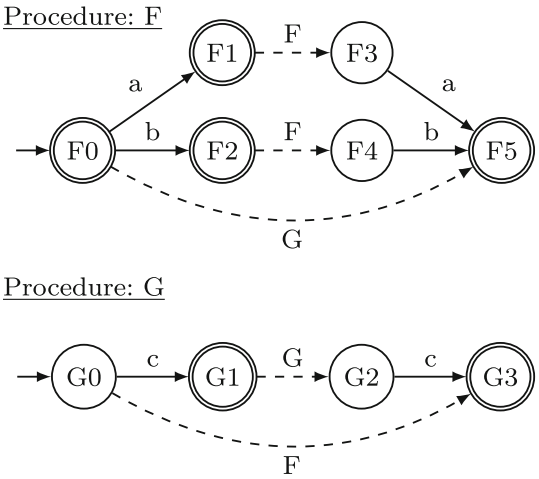

Fig. 6 Inferred (DFA-based) SPA model for the palindrome language. Sink states and corresponding transitions of the DFAs are omitted for readability

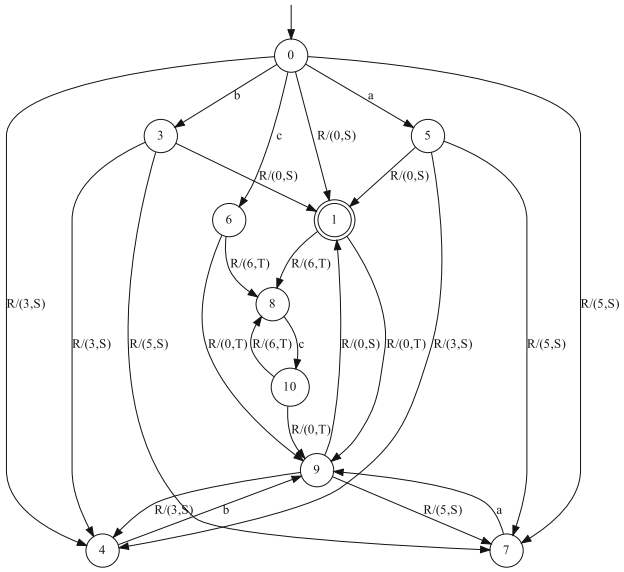

Fig. 7 Inferred 1-SEVPA model for the palindrome language. Call transitions have been omitted, because in a 1-SEVPA all call transitions lead into the single module entry location. Return transitions are labeled with their required stack contents. Furthermore, the sink state (including all return transitions leading into the sink state) have been omitted

sition every possible stack symbol needs to be regarded. This results in a lot of overhead for determining transition successors. While in general this allows one to capture more complex behavior (e.g. returns to different procedures/modules), it shows no benefit in our context of procedural systems revolving around the copy-rule semantics.

We have observed similar results for other applications, such as inferring the structure of XML documents [17] or exponential systems which try to approximate recursive systems with pure regular automata. See https://github.com/LearnLib/learnlib-spa for further benchmark data. We plan to further analyze and exploit these characteristics for complex, large-scale systems in the future.

9 Conclusion and future work

In this paper we have presented a compositional approach for active automata learning of Systems of Procedural Automata (SPAs), an extension of Deterministic Finite Automata (DFAs) to systems of DFAs that can mutually call each other. SPAs are of high practical relevance, as they allow one to efficiently learn intuitive, recursive models of recursive programs after an easy instrumentation that makes calls and returns observable. Instrumentations like this are a very fruitful example of how to exploit additional (architectural) knowledge during the learning process in order to boost performance. In this case, they even expand the reach of regular active automata learning to cover all context-free languages and this without increasing the required query complexity. This is possible because the learning process for SPAs can be organized as a simultaneous inference of individual DFAs for each of the involved procedures via projection and expansion that bridge the gap between the global view concerning the SPA and the local views for the individual procedural automata.

There are numerous directions for future work: The treatment of equivalence queries—the drivers of the learning

Table 1 Performance measures of individual learning setups. The averages (ø) and standard deviation (*sd*) of 15 runs are presented. "CE" is short for *counterexample*, "MQ" is short for *membership query* and *symbols* refer to the aggregated number of symbols of all membership queries

		No. of CEs		No. of MQs		No. of symbols		Hyp. size
		ø	*sd*	ø	*sd*	ø	*sd*	ø
SPA	L* (classic) [5]	7.4	0.7	435.5	41.5	3794.7	427.9	12.0
	L* (Rivest & Schapire) [44]	8.7	0.5	315.2	10.5	2356.1	83.0	12.0
	L* (Kearns & Vazirani) [32]	9.2	0.4	214.1	10.7	1457.8	110.3	12.0
	DiscriminationTree [21]	9.1	0.5	218.1	10.3	1452.5	109.8	12.0
	TTT [29]	9.3	0.6	186.6	7.4	1265.6	88.2	12.0
VPA	DiscriminationTree	7.9	0.4	1616.5	2.5	15176.7	281.7	11.0
	TTT	8.1	0.7	1651.9	93.3	15343.1	703.4	11.0

process—is a research topic of its own. They are typically approximated using membership queries and often depend on application-specific heuristics to be effective. A common heuristic involves model checking to generate membership queries for detecting potential counterexamples. This works particularly well if some assumed behavioral properties are known at learning time. That this approach can also be applied for procedural systems has been shown in [11]. Especially in the context of procedural systems, where errors occur locally within procedures, counterexamples generated using fuzzing [18,19,37] look promising to have a positive impact on the performance of the learning process.

An alternative method to realize equivalence queries is a change of perspective in the direction of never-stop or life-long learning [7]. The underlying main idea is to instrument the (potentially in-production) system with a monitoring mechanism that observes and controls its runs on the basis of previously learned hypothesis models. Whenever the monitor recognizes a discrepancy between the current hypothesis model and the system, the corresponding trace is fed to the learner in order to refine the hypothesis model and the corresponding monitor. Subsequently, the life-long learning process continues with the next monitoring phase. This approach, which is characterized by its never-stopping, user-driven counterexample search, has shown promising results in a number of software projects in the past [7,31,40,47]. Life-long learning comes with a challenge: counterexamples may be excessively long, as they typically arise as unexpected continuations of days-long normal operating. "Classical" AAL algorithms are not able to deal with this characteristic as their complexity depends (a least) linearly on the length of counterexamples. In our current research we are observing that the procedural structure of SPAs with their potential to dynamically optimize access, terminating and return sequences as well as using redundancy-free learners (e.g. TTT [29]) as procedural learners have a lot of potential to tackle these issues and allow for *practical* context-free life-long learning.

Finally, SPAs provide a very powerful basis for further conceptual extension. A particularly interesting challenge is how far further system properties like inputs/outputs (e.g. Mealy machines) or other data-flow properties (e.g. register automata [22]) can be married with our procedural approach. Currently, the context-free nature of SPAs does not allow individual procedures to account for different execution contexts (compared to e.g. VPAs which, however, pay for this expressiveness with increased complexity). This may be tackled by the concept of *call abstraction* where different procedures (representing context-dependent behavior) may share the same call symbol. Here, concepts such as *alphabet abstraction refinement* [23,27] may prove helpful to create a dynamically adjusting learning approach that is as simple as possible but as specific as necessary, combining performance with expressive power. The concept of instrumentation may turn out to be a powerful enabler in this context.

Funding Open Access funding enabled and organized by Projekt DEAL.

Appendix: Proofs

Lemma 2 *Let Σ be an SPA alphabet, $\widehat{w} \in (\widehat{\Sigma}_{call} \cup \widehat{\Sigma}_{int})^*$ and $\sigma \in ST(\Gamma)$.*

$$(\widehat{w} \cdot \widehat{r}, \sigma) \xrightarrow{z}{}^*(\widehat{r}, \sigma) \Rightarrow \alpha(z) = \widehat{w}$$

Proof This follows by induction over the length of $\widehat{w}$.

- For $\widehat{w} = \varepsilon$, we have $(\widehat{w} \cdot \widehat{r}, \sigma) = (\widehat{r}, \sigma)$ and therefore $z = \varepsilon$, hence $\alpha(z) = \varepsilon = \widehat{w}$.
- Now let $\widehat{w} = \widehat{u} \cdot \widehat{v}$, with $\widehat{u} \in (\widehat{\Sigma}_{call} \cup \widehat{\Sigma}_{int})$, $\widehat{v} \in (\widehat{\Sigma}_{call} \cup \widehat{\Sigma}_{int})^*$ and $y \in \Sigma^*$ such that

  $$(\widehat{v} \cdot \widehat{r}, \sigma) \xrightarrow{y}{}^*(r, \sigma) \Rightarrow \alpha(y) = \widehat{v}$$

  holds. We distinguish whether $\widehat{u}$ is a call symbol or an internal symbol. If $\widehat{u} \in \widehat{\Sigma}_{call}$, the SPA will emit $x \in WM(\Sigma)$ consisting of u, possibly a well-matched sub-word, and a matching (to u) return symbol r before reaching the $(\widehat{v} \cdot \widehat{r}, \sigma)$ configuration. We have

  $$(\widehat{u} \cdot \widehat{v} \cdot \widehat{r}, \sigma) \xrightarrow{x}{}^*(\widehat{v} \cdot \widehat{r}, \sigma) \xrightarrow{y}{}^*(\widehat{r}, \sigma).$$

  Since x is well-matched, α will map x to $\widehat{u}$ and continue to process y. Hence

  $$\alpha(z) = \alpha(x \cdot y) = \widehat{u} \cdot \alpha(y) = \widehat{u} \cdot \widehat{v} = \widehat{w}.$$

  If $\widehat{u} \in \widehat{\Sigma}_{int}$, the SPA will emit u (int-rule) and α will map this symbol to $\widehat{u}$. We have

  $$(\widehat{u} \cdot \widehat{v} \cdot \widehat{r}, \sigma) \xrightarrow{u} (\widehat{v} \cdot \widehat{r}, \sigma) \xrightarrow{y}{}^*(\widehat{r}, \sigma).$$

  Hence, $\alpha(z) = \alpha(u \cdot y) = \widehat{u} \cdot \alpha(y) = \widehat{u} \cdot \widehat{v} = \widehat{w}$. □

Lemma 3 *Let Σ be an SPA alphabet, S be an SPA over Σ and $w \in WM(\Sigma)$.*

$$\forall c \in \Sigma_{call} : \forall ts \in TS_c : \alpha(ts) \in L(P^c)$$

Proof Let $c \in \Sigma_{call}$ and $ts \in TS_c$ be arbitrary. Since ts is part of an accepted word, we know that there exists a path in the SOS transition system such that

$$(\widehat{c} \cdot \widehat{v}, \sigma) \xrightarrow{c}{}^{*} (\widehat{w} \cdot \widehat{r}, \widehat{v} \bullet \sigma) \xrightarrow{ts}{}^{*} (\widehat{r}, \widehat{v} \bullet \sigma)$$

for some $\widehat{v} \in \widehat{\Sigma}^*$, $\widehat{w} \in (\widehat{\Sigma}_{call} \cup \widehat{\Sigma}_{int})^*$, $\sigma \in ST(\Gamma)$. According to the definition of call-rules we have $\widehat{w} \in L(P^c)$ and by Lemma 2 $\alpha(ts) = \widehat{w}$. Hence, $\alpha(ts) \in L(P^c)$. □

Theorem 1 (Localization theorem) *Let Σ be an SPA alphabet, S be an SPA over Σ and $w \in WM(\Sigma)$ be rooted in c_0.*

$$w \in L(S) \Leftrightarrow$$
$$\forall (c, i) \in Inst_w : \alpha(w[i+1, \rho_w(i+1) - 1]) \in L(P^c)$$

Proof Let $w \in WM(\Sigma)$ be rooted in c_0.
$\Rightarrow$: Let $w \in L(S)$ and $(c, i) \in Inst_w$ be arbitrary. By Definition 10 we know that there exists a $ts \in TS_c$ with $ts = w[i+1, \rho_w(i+1) - 1]$. Lemma 3 then directly concludes the statement.
$\Leftarrow$: We show via contraposition that

$$w \notin L(S) \Rightarrow$$
$$\exists (c, i) \in Inst_w : \alpha(w[i+1, \rho_w(i+1) - 1]) \notin L(P^c)$$

Let $w = u_1 \cdot a \cdot u_2$ for some $u_1, u_2 \in \Sigma^*, a \in \Sigma$. Then there exist $\widehat{v}_1 \in \widehat{\Sigma}^*, \sigma_1 \in ST(\Gamma)$ such that

$$(\widehat{c}_0, \bot) \xrightarrow{u_1}{}^{*} (\widehat{v}_1, \sigma_1)$$

but $\nexists\, \widehat{v}_2 \in \Sigma^*, \sigma_2 \in ST(\Gamma)$ such that

$$(\widehat{c}_0, \bot) \xrightarrow{u_1 \cdot a}{}^{*} (\widehat{v}_2, \sigma_2).$$

Such a decomposition has to exist because otherwise (i.e. $u_1 \cdot a = w, \widehat{v}_2 = \varepsilon, \sigma_2 = \bot$) w would be in the language of the SPA, contradicting our assumption. Combined, this means $\nexists\, \widehat{v}_2 \in \widehat{\Sigma}^*, \sigma_2 \in ST(\Gamma)$ such that

$$(\widehat{v}_1, \sigma_1) \xrightarrow{a} (\widehat{v}_2, \sigma_2).$$

Let $(c_*, i_*) \in Inst_w$ such that $i_* \leq |u_1|$ is the largest index for which u_1 is a prefix of $w[, \rho_w(i_* + 1) - 1]$, i.e. it is procedure c_* that cannot emit the violating action a. We then have $\alpha(w[i_* + 1, \rho_w(i_* + 1) - 1]) \notin L(P^{c_*})$ as required. □

Corollary 1 (Membership query expansion) *Let Σ be an SPA alphabet and S be an SPA over Σ.*

$$\widehat{w} \in L(P^c) \Leftrightarrow as \cdot \gamma(\widehat{w}) \cdot rs \in L(S)$$

for all $c \in \Sigma_{call}$, $(as, rs) \in Cont_c$.

Proof By Definition 10 it is guaranteed that

$$(\widehat{c}_0, \bot) \xrightarrow{as}{}^{*} (\widehat{w} \cdot \widehat{r}, \sigma)$$

and

$$(\widehat{r}, \sigma) \xrightarrow{rs}{}^{*} (\varepsilon, \bot)$$

for all $\widehat{w} \in L(P^c)$ and a fixed (depending on as) $\sigma \in ST(\Gamma)$. What remains to be shown is

$$(\widehat{w} \cdot \widehat{r}, \sigma) \xrightarrow{\gamma(\widehat{w})}{}^{*} (\widehat{r}, \sigma).$$

This follows by induction over the length of $\widehat{w}$.

- If $\widehat{w} = \varepsilon$ we have $\gamma(\widehat{w}) = \varepsilon$ and the statement follows.
- Now let $\widehat{w} = \widehat{u} \cdot \widehat{v}$ with $\widehat{u} \in (\widehat{\Sigma}_{call} \cup \widehat{\Sigma}_{int})$, $\widehat{v} \in (\widehat{\Sigma}_{call} \cup \widehat{\Sigma}_{int})^*$ such that

  $$(\widehat{v} \cdot \widehat{r}, \sigma) \xrightarrow{\gamma(\widehat{v})}{}^{*} (\widehat{r}, \sigma).$$

  We distinguish whether $\widehat{u}$ is a call symbol or an internal symbol. If $\widehat{u} \in \widehat{\Sigma}_{call}$ we have $\gamma(\widehat{u}) = u \cdot ts[u] \cdot r$. Since $ts[u]$ is a terminating sequence (i.e. part of an accepted word), we know that there exists an $\widehat{x} \in L(P^u)$ such that

  $$(\widehat{u} \cdot \widehat{v} \cdot \widehat{r}, \sigma) \xrightarrow{u} (\widehat{x} \cdot \widehat{r}, \widehat{v} \cdot \widehat{r} \bullet \sigma) \xrightarrow{ts[u]}{}^{*} (\widehat{r}, \widehat{v} \cdot \widehat{r} \bullet \sigma) \xrightarrow{r} (\widehat{v} \cdot \widehat{r}, \sigma).$$

  and therefore

  $$(\widehat{u} \cdot \widehat{v} \cdot \widehat{r}, \sigma) \xrightarrow{\gamma(\widehat{u})}{}^{*} (\widehat{v} \cdot \widehat{r}, \sigma) \xrightarrow{\gamma(\widehat{v})}{}^{*} (\widehat{r}, \sigma)$$

  which is equivalent to

  $$(\widehat{u} \cdot \widehat{v} \cdot \widehat{r}, \sigma) \xrightarrow{\gamma(\widehat{u} \cdot \widehat{v})}{}^{*} (\widehat{r}, \sigma)$$

  since $\gamma(\widehat{u}) \cdot \gamma(\widehat{v}) = \gamma(\widehat{u} \cdot \widehat{v})$. If $\widehat{u} \in \widehat{\Sigma}_{int}$ we have $\gamma(\widehat{u}) = u$ and by application of an int-rule, we have

  $$(\widehat{u} \cdot \widehat{v} \cdot \widehat{r}, \sigma) \xrightarrow{\gamma(\widehat{u})} (\widehat{v} \cdot \widehat{r}, \sigma) \xrightarrow{\gamma(\widehat{v})}{}^{*} (\widehat{r}, \sigma)$$

  and the statement directly follows since $\gamma(\widehat{u}) \cdot \gamma(\widehat{v}) = u \cdot \gamma(\widehat{v}) = \gamma(\widehat{u} \cdot \widehat{v})$. □

Lemma 4 *Let Σ be an SPA alphabet and $w \in WM(\Sigma)$ be a rooted word. Then any decomposition of $w = u \cdot v$ can be written as*

$$\begin{aligned} u &= c_{i_1} \cdot w_1 \cdot \ldots \cdot c_{i_j} \cdot w_j \\ v &= w_{j+1} \cdot r \cdot w_{j+2} \cdot \ldots \cdot r \end{aligned}$$

with $w_i \in WM(\Sigma)$.

Proof This directly follows from the definition of rooted (i.e. well-matched) words, where for every prefix u, $\beta(u) \geq 0$ and for every suffix v, $\beta(v) \leq 0$ holds. $\beta(u)$ gives the "nesting depth" of w after parsing u and corresponds to the number of unmatched call symbols $c_{i_1}, c_{i_2}, \ldots$ at that position. Analogously, one can isolate from the suffix v the unmatched return symbols. □

Lemma 5 *Let Σ be an SPA alphabet and $w, w_1, w_2 \in WM(\Sigma)$ be well-matched words.*

$$[\![w_1]\!] \cdot [\![w_2]\!] = [\![w_1 \cdot w_2]\!] \tag{1}$$
$$[\![[\![w]\!]]\!] = [\![w]\!] \tag{2}$$

Proof (1): Let $w_1, w_2 \in WM(\Sigma)$ be arbitrary. We have

$$\begin{aligned} [\![w_1]\!] \cdot [\![w_2]\!] &= \gamma(\alpha(w_1)) \cdot \gamma(\alpha(w_2)) && (3)\\ &= \gamma(\alpha(w_1) \cdot \alpha(w_2)) && (4)\\ &= \gamma(\alpha(w_1 \cdot w_2)) && (5)\\ &= [\![w_1 \cdot w_2]\!] && (6) \end{aligned}$$

Equation 5 holds because w_1 is a well-matched word and thus $\forall (c, i) \in Inst_{w_1} \colon \rho_{w_1[i+1;]}(1) \leq |w_1|$. This guarantees that the α projection does not process symbols of w_2 until all symbols of w_1 have been processed and therefore α concatenates the individual projections.
(2): This follows by structural induction over w (cf. Definition 2).

- Let $w \in \Sigma_{int}^*$. For any internal symbol $i \in \Sigma_{int}$, we have $[\![i]\!] = \gamma(\alpha(i)) = \gamma(\widehat{i}) = i$, i.e. $[\![\cdot]\!]$ coincides with the identity function. Thus, $w = [\![w]\!] = [\![[\![w]\!]]\!]$ and the statement directly follows.
- Let $w \in WM(\Sigma)$ such that $w = c \cdot v \cdot r$ for some $c \in \Sigma_{call}$, $v \in WM(\Sigma)$. We have

$$\begin{aligned} [\![c \cdot v \cdot r]\!] &= \gamma(\alpha(c \cdot v \cdot r)) && (7)\\ &= \gamma(\widehat{c}) && (8)\\ &= \gamma(\alpha(c \cdot ts[c] \cdot r)) && (9)\\ &= \gamma(\alpha(\gamma(\widehat{c}))) && (10)\\ &= \gamma(\alpha(\gamma(\alpha(c \cdot v \cdot r)))) && (11)\\ &= [\![[\![c \cdot v \cdot r]\!]]\!] && (12) \end{aligned}$$

- Let $w \in WM(\Sigma)$ such that $w = w_1 \cdot w_2$ for some $w_1, w_2 \in WM(\Sigma)$ and let the induction hypothesis hold for w_1, w_2. We have

$$\begin{aligned} [\![w_1 \cdot w_2]\!] &= [\![w_1]\!] \cdot [\![w_2]\!] && (13)\\ &= [\![[\![w_1]\!]]\!] \cdot [\![[\![w_2]\!]]\!] && (14)\\ &= [\![[\![w_1]\!] \cdot [\![w_2]\!]]\!] && (15)\\ &= [\![[\![w_1 \cdot w_2]\!]]\!] && (16) \end{aligned}$$

Equation 13 and the subsequent ones hold because w_1, w_2 are well-matched and hence Eq. 1 is applicable. Equation 14 holds by induction hypothesis.

□

Lemma 6 *Let Σ be an SPA alphabet, S be a ts-conform SPA over Σ. Then we have for all $w \in WM(\Sigma)$ and $w_c \in (\Sigma_{call} \cdot WM(\Sigma))^*$*

$$(\widehat{s}_1, \sigma) \xrightarrow{w}{}^*(\widehat{s}_2, \sigma) \Rightarrow (\widehat{s}_1, \sigma) \xrightarrow{[\![w]\!]}{}^*(\widehat{s}_2, \sigma) \tag{17}$$
$$(\widehat{s}_1, \sigma_1) \xrightarrow{w_c}{}^*(\widehat{s}_2, \sigma_2) \Rightarrow (\widehat{s}_1, \sigma_1) \xrightarrow{[\![w_c]\!]^*}{}^*(\widehat{s}_2, \sigma_2) \tag{18}$$

for some $\widehat{s}_1, \widehat{s}_2 \in \widehat{\Sigma}^*, \sigma, \sigma_1, \sigma_2 \in ST(\Gamma)$.

Proof (17): This follows by structural induction over w (cf. Definition 2).

- Let $w \in \Sigma_{int}^*$. As argued in the proof of Lemma 5, $[\![\cdot]\!]$ coincides with the identity function for internal symbols. Thus, $w = [\![w]\!]$ and the statement directly follows.
- Let $w \in WM(\Sigma)$ such that $w = c \cdot v \cdot r$ for some $c \in \Sigma_{call}$, $v \in WM(\Sigma)$. By premise, we have

$$(\widehat{s}_1, \sigma) \xrightarrow{w}{}^*(\widehat{s}_2, \sigma)$$

and by Definition 9

$$(\widehat{s}_1, \sigma) \xrightarrow{c} (\widehat{t}_1, \psi) \xrightarrow{v}{}^*(\widehat{t}_2, \psi) \xrightarrow{r} (\widehat{s}_2, \sigma)$$

for some $\widehat{t}_1, \widehat{t}_2 \in \widehat{\Sigma}^*, \psi \in ST(\Gamma)$. For any rooted word $c \cdot v \cdot r \in WM(\Sigma)$ we have

$$[\![c \cdot v \cdot r]\!] = \gamma(\alpha(c \cdot v \cdot r)) = \gamma(\widehat{c}) = c \cdot ts[c] \cdot r.$$

Since S is ts-conform, we know that there exists a $\widehat{v} \in L(P^c)$ such that

$$(\widehat{s}_1, \sigma) \xrightarrow{c} (\widehat{v} \cdot r, \psi) \xrightarrow{ts[c]}{}^*(\widehat{r}, \psi) \xrightarrow{r} (\widehat{s}_2, \sigma)$$

for some $\psi \in ST(\Gamma)$ and hence

$$(\widehat{s}_1, \sigma) \xrightarrow{[\![w]\!]}{}^*(\widehat{s}_2, \sigma).$$

– Let $w \in WM(\Sigma)$ such that $w = w_1 \cdot w_2$ for some $w_1, w_2 \in WM(\Sigma)$ and let the induction hypothesis hold for w_1, w_2. By premise, we have

$$(\widehat{s}_1, \sigma) \xrightarrow{w}{}^*(\widehat{s}_2, \sigma)$$

and by Definition 9

$$(\widehat{s}_1, \sigma) \xrightarrow{w_1}{}^*(\widehat{t}_1, \sigma) \xrightarrow{w_2}{}^*(\widehat{s}_2, \sigma)$$

for some $\widehat{t}_1 \in \widehat{\Sigma}^*$. By induction hypothesis, we have

$$(\widehat{s}_1, \sigma) \xrightarrow{[\![w_1]\!]}{}^*(\widehat{t}_1, \sigma) \xrightarrow{[\![w_2]\!]}{}^*(\widehat{s}_2, \sigma)$$

and by application of Lemma 5 (1)

$$(\widehat{s}_1, \sigma) \xrightarrow{[\![w_1 \cdot w_2]\!]}{}^*(\widehat{s}_2, \sigma).$$

(18): This follows by induction over the number n of unmatched call symbols in w_c.

– For $n = 0$ we have $\varepsilon = w_c \in (\Sigma_{call} \cdot WM(\Sigma))^0$ and $[\![w_c]\!]^* = \varepsilon$. Here, the statement directly follows.
– For the induction step let $w_c \in (\Sigma_{call} \cdot WM(\Sigma))^{n+1}$ such that $w_c = w_{1c} \cdot c_{i_{n+1}} \cdot w_2$ with $w_{1c} \in (\Sigma_{call} \cdot WM(\Sigma))^n$, $w_2 \in WM(\Sigma)$ and let the statement hold for w_{1c}. By premise of the statement for $n+1$ we have

$$(\widehat{s}_1, \sigma_1) \xrightarrow{w_c}{}^*(\widehat{s}_2, \sigma_2)$$

and by Definition 9

$$(\widehat{s}_1, \sigma_1) \xrightarrow{w_{1c}}{}^*(\widehat{t}_1, \psi_1) \xrightarrow{c_{i_{n+1}}} (\widehat{t}_2, \psi_2) \xrightarrow{w_2}{}^*(\widehat{s}_2, \sigma_2)$$

for some $\widehat{t}_1, \widehat{t}_2 \in \widehat{\Sigma}^*$, $\psi_1, \psi_2 \in ST(\Gamma)$. By applying the induction hypothesis for w_{1c} we have

$$(\widehat{s}_1, \sigma_1) \xrightarrow{[\![w_{1c}]\!]^*}{}^*(\widehat{t}_1, \psi_1) \xrightarrow{c_{i_{n+1}}} (\widehat{t}_2, \psi_2) \xrightarrow{w_2}{}^*(\widehat{s}_2, \sigma_2)$$

and by applying (1) we have

$$(\widehat{s}_1, \sigma_1) \xrightarrow{[\![w_{1c}]\!]^*}{}^*(\widehat{t}_1, \psi_1) \xrightarrow{c_{i_{n+1}}} (\widehat{t}_2, \psi_2) \xrightarrow{[\![w_2]\!]}{}^*(\widehat{s}_2, \sigma_2).$$

Using Definition 14 to decompose $[\![w_{1c}]\!]^*$ and then (re-) compose the individual sub-words and call symbols including $c_{i_{n+1}}$ and w_2, we can conclude

$$(\widehat{s}_1, \sigma_1) \xrightarrow{[\![w_{1c} \cdot c_{i_{n+1}} \cdot w_2]\!]^*}{}^*(\widehat{s}_2, \sigma_2)$$

which concludes the statement for $n+1$.

This induction shows that the statement holds for all possible values of n and therefore especially for arbitrary $w_c \in (\Sigma_{call} \cdot WM(\Sigma))^* = \bigcup_{i=0}^{\infty}(\Sigma_{call} \cdot WM(\Sigma))^i$. □

Lemma 7 *Let Σ be an SPA alphabet, S be a ts-conform SPA over Σ and $w \in WM(\Sigma)$ rooted with $|Inst_w| \geq 2$. Let r_h denote the h-th return symbol index of w such that $r_h < r_{h+1}$ for all $h \in \{1, \ldots, |Inst_w| - 1\}$.*

$$[\![w[, r_h]]\!]^* \cdot w[r_h + 1,] \in L(S) \Rightarrow$$
$$[\![w[, r_{h+1}]]\!]^* \cdot w[r_{h+1} + 1,] \in L(S)$$

Proof Let $[\![w[, r_h]]\!]^* \cdot w[r_h + 1,] \in L(S)$. This means there exists a path in the SOS transition system such that

$$\begin{aligned}(\widehat{c}_0, \bot) &\xrightarrow{[\![w[, r_h]]\!]^*}{}^* && (\widehat{s}_1, \sigma_1)\\ &\xrightarrow{w[r_h+1, r_{h+1}]}{}^* && (\widehat{s}_2, \sigma_2)\\ &\xrightarrow{w[r_{h+1}+1,]}{}^* && (\varepsilon, \bot)\end{aligned}$$

for some $\widehat{s}_1, \widehat{s}_2 \in \widehat{\Sigma}^*, \sigma_1, \sigma_2 \in ST(\Gamma)$. $w[, r_h]$ is a prefix of the rooted word w and therefore can be decomposed according to Lemma 4 as

$$w[, r_h] = c_{j_1} \cdot w_1 \cdot \ldots \cdot c_{j_m} \cdot w_m$$

for some $w_j \in WM(\Sigma)$. Hence

$$\begin{aligned}[\![w[, r_h]]\!]^* &= [\![c_{j_1} \cdot w_1 \cdot \ldots \cdot c_{j_m} \cdot w_m]\!]^*\\ &= c_{j_1} \cdot [\![w_1]\!] \cdot \ldots \cdot c_{j_m} \cdot [\![w_m]\!].\end{aligned}$$

The word $w[r_h + 1, r_{h+1}]$ consists of an arbitrary (potentially empty) sequence of internal and call symbols and ends with a single return symbol $w[r_{h+1}]$. In the following, we distinguish the two cases, where $w[r_h + 1, r_{h+1}]$ contains no call symbols and at least one call symbol.

– If $w[r_h + 1, r_{h+1}]$ contains no call symbols, we have

$$w[r_h + 1, r_{h+1}] = w_{m_2} \cdot r$$

for some $w_{m_2} \in \Sigma_{int}^*$ and

$$\begin{aligned}&w[, r_h] \cdot w[r_h + 1, r_{h+1}]\\ &\quad = w[, r_{h+1}]\\ &\quad = c_{j_1} \cdot w_1 \cdot \ldots \cdot c_{j_{m-1}} \cdot v\end{aligned}$$

with $v \in WM(\Sigma)$ such that

$$v = w_{j_{m-1}} \cdot c_{j_m} \cdot w_m \cdot w_{m_2} \cdot r.$$

If r_{h+1} is the last return symbol index of w, we have $w = v$. Since $[\![\cdot]\!]^*$ preserves unmatched call symbols, we

have

$$[\![w[,r_h]]\!]^* \cdot w[r_h+1,r_{h+1}] = c_{j_1} \cdot [\![w_1]\!] \cdot \ldots \cdot u$$

for some $u \in WM(\Sigma)$ with

$$u = [\![w_{j_{m-1}}]\!] \cdot c_{j_m} \cdot [\![w_m]\!] \cdot w_{m_2} \cdot r.$$

This directly gives

$$[\![w[,r_h]]\!]^* \cdot w[r_h+1,r_{h+1}] \in (\Sigma_{call} \cdot WM(\Sigma))^*$$

or

$$[\![w[,r_h]]\!]^* \cdot w[r_h+1,r_{h+1}] \in WM(\Sigma)$$

if r_{h+1} is the last return symbol index. By Lemma 6 (18) we have

$$(\widehat{c}_0, \bot) \xrightarrow{[\![[\![w[,r_h]]\!]^* \cdot w[r_h+1,r_{h+1}]]\!]^*} {}^*(\widehat{s}_2, \sigma_2).$$

We show with Lemma 5

$$\begin{aligned}
&[\![v]\!] \\
&= [\![w_{j_{m-1}} \cdot c_{j_m} \cdot w_m \cdot w_{m_2} \cdot r]\!] \\
&= [\![w_{j_{m-1}}]\!] \cdot [\![c_{j_m} \cdot w_m \cdot w_{m_2} \cdot r]\!] \\
&= [\![[\![w_{j_{m-1}}]\!]]\!] \cdot \gamma(\alpha(c_{j_m} \cdot w_m \cdot w_{m_2} \cdot r)) \\
&= [\![[\![w_{j_{m-1}}]\!]]\!] \cdot \gamma(\widehat{c}_{j_m}) \\
&= [\![[\![w_{j_{m-1}}]\!]]\!] \cdot \gamma(\alpha(c_{j_m} \cdot [\![w_m]\!] \cdot w_{m_2} \cdot r)) \\
&= [\![[\![w_{j_{m-1}}]\!]]\!] \cdot [\![c_{j_m} \cdot [\![w_m]\!] \cdot w_{m_2} \cdot r]\!] \\
&= [\![[\![w_{j_{m-1}}]\!] \cdot c_{j_m} \cdot [\![w_m]\!] \cdot w_{m_2} \cdot r]\!] \\
&= [\![u]\!]
\end{aligned}$$

Furthermore, we have

$$\begin{aligned}
&[\![[\![w[,r_h]]\!]^* \cdot w[r_h+1,r_{h+1}]]\!]^* \\
&= [\![[\![c_{j_1} \cdot w_1 \cdot \ldots \cdot c_{j_m} \cdot w_m]\!]^* \cdot w_{m_2} \cdot r]\!]^* \\
&= [\![c_{j_1} \cdot [\![w_1]\!] \cdot \ldots \cdot c_{j_m} \cdot [\![w_m]\!] \cdot w_{m_2} \cdot r]\!]^* \\
&= [\![c_{j_1} \cdot [\![w_1]\!] \cdot \ldots \cdot u]\!]^* \\
&= c_{j_1} \cdot [\![[\![w_1]\!]]\!] \cdot \ldots \cdot [\![u]\!] \\
&= c_{j_1} \cdot [\![w_1]\!] \cdot \ldots \cdot [\![v]\!] \\
&= [\![c_{j_1} \cdot w_1 \cdot \ldots \cdot v]\!]^* \\
&= [\![w[,r_{h+1}]]\!]^*
\end{aligned}$$

This shows the existence of the path

$$(\widehat{c}_0, \bot) \xrightarrow{[\![w[,r_{h+1}]]\!]^*} {}^*(\widehat{s}_2, \sigma_2) \xrightarrow{w[r_{h+1}+1,]} {}^*(\varepsilon, \bot)$$

and therefore $[\![w[,r_{h+1}]]\!]^* \cdot w[r_{h+1}+1,] \in L(S)$.

– If $w[r_h+1,r_{h+1}]$ contains at least one (or $n-m$) call symbol(s), we have

$$w[r_h+1,r_{h+1}] = w_{m_2} \cdot c_{j_{m+1}} \cdot w_{m+1} \cdot \ldots \cdot c_{j_n} \cdot w_n$$

for some $w_j \in WM(\Sigma)$ and

$$\begin{aligned}
&w[,r_h] \cdot w[r_h+1,r_{h+1}] \\
&\quad = w[,r_{h+1}] \\
&\quad = c_{j_1} \cdot w_1 \cdot \ldots \cdot c_{j_m} \cdot w_m \cdot w_{m_2} \cdot \ldots \cdot c_{j_n} \cdot w_n
\end{aligned}$$

Since $[\![\cdot]\!]^*$ preserves unmatched call symbols, we have

$$\begin{aligned}
&[\![w[,r_h]]\!]^* \cdot w[r_h+1,r_{h+1}] \\
&\quad = c_{j_1} \cdot [\![w_1]\!] \cdot \ldots \cdot c_{j_m} \cdot [\![w_m]\!] \cdot w_{m_2} \cdot \ldots \cdot c_{j_n} \cdot w_n
\end{aligned}$$

This directly gives

$$[\![w[,r_h]]\!]^* \cdot w[r_h+1,r_{h+1}] \in (\Sigma_{call} \cdot WM(\Sigma))^*$$

and by Lemma 6 (18)

$$(\widehat{c}_0, \bot) \xrightarrow{[\![[\![w[,r_h]]\!]^* \cdot w[r_h+1,r_{h+1}]]\!]^*} {}^*(\widehat{s}_2, \sigma_2).$$

We show with Lemma 5

$$\begin{aligned}
&[\![[\![w[,r_h]]\!]^* \cdot w[r_h+1,r_{h+1}]]\!]^* \\
&= [\![[\![c_{j_1} \cdot w_1 \cdot \ldots \cdot w_m]\!]^* \cdot w_{m_2} \cdot \ldots \cdot c_{j_n} \cdot w_n]\!]^* \\
&= [\![c_{j_1} \cdot [\![w_1]\!] \cdot \ldots \cdot [\![w_m]\!] \cdot w_{m_2} \cdot \ldots \cdot c_{j_n} \cdot w_n]\!]^* \\
&= c_{j_1} \cdot [\![[\![w_1]\!]]\!] \cdot \ldots \cdot [\![[\![w_m]\!]]\!] \cdot [\![w_{m_2}]\!] \cdot \ldots \cdot c_{j_n} \cdot [\![w_n]\!] \\
&= c_{j_1} \cdot [\![w_1]\!] \cdot \ldots \cdot [\![w_m]\!] \cdot [\![w_{m_2}]\!] \cdot \ldots \cdot c_{j_n} \cdot [\![w_n]\!] \\
&= c_{j_1} \cdot [\![w_1]\!] \cdot \ldots \cdot [\![w_m \cdot w_{m_2}]\!] \cdot \ldots \cdot c_{j_n} \cdot [\![w_n]\!] \\
&= [\![c_{j_1} \cdot w_1 \cdot \ldots w_m \cdot w_{m_2} \cdot \ldots \cdot c_{j_n} \cdot w_n]\!]^* \\
&= [\![w[,r_{h+1}]]\!]^*
\end{aligned}$$

This shows the existence of the path

$$(\widehat{c}_0, \bot) \xrightarrow{[\![w[,r_{h+1}]]\!]^*} {}^*(\widehat{s}_2, \sigma_2) \xrightarrow{w[r_{h+1}+1,]} {}^*(\varepsilon, \bot)$$

and therefore $[\![w[,r_{h+1}]]\!]^* \cdot w[r_{h+1}+1,] \in L(S)$.

Thus, in both cases the statement holds. □

Theorem 2 (Acceptance monotonicity of $[\![\cdot]\!]^*$) *Let Σ be an SPA alphabet, S be a ts-conform SPA over Σ, $w \in WM(\Sigma)$ be rooted and r_h, r_k be indices of return symbols of w with $r_h < r_k$. Then we have*

$$\begin{aligned}
&[\![w[,r_h]]\!]^* \cdot w[r_h+1,] \in L(S) \Rightarrow \\
&\quad [\![w[,r_k]]\!]^* \cdot w[r_k+1,] \in L(S)
\end{aligned}$$

Proof This directly follows by repeated application of Lemma 7 and the transitivity of the logical implication. □

References

1. Aarts, F., Fiterau-Brostean, P., Kuppens, H., Vaandrager, F.: Learning register automata with fresh value generation. In: Leucker, M., Rueda, C., Valencia, F.D. (eds.) Theoretical Aspects of Computing-ICTAC 2015, pp. 165–183. Springer, Cham (2015)
2. Alur, R., Etessami, K., Yannakakis, M.: Analysis of recursive state machines. In: Berry, G., Comon, H., Finkel, A. (eds.) Computer Aided Verification: 13th International Conference, pp. 207–220. Springer, Berlin (2001). https://doi.org/10.1007/3-540-44585-4_18
3. Alur, R., Kumar, V., Madhusudan, P., Viswanathan, M.: Congruences for visibly pushdown languages. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) Automata, Languages and Programming: 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11–15, 2005. Proceedings, pp. 1102–1114. Springer, Berlin (2005). https://doi.org/10.1007/11523468_89
4. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Proceedings of the 36th Annual ACM Symposium on Theory of Computing, STOC'04, p. 202–211. Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/1007352.1007390
5. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87–106 (1987)
6. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press, Cambridge (2008)
7. Bertolino, A., Calabrò, A., Merten, M., Steffen, B.: Never-stop learning: continuous validation of learned models for evolving systems through monitoring. ERCIM News **2012**(88), 28–29 (2012)
8. Bollig, B., Habermehl, P., Leucker, M., Monmege, B.: A fresh approach to learning register automata. In: Béal, M.P., Carton, O. (eds.) Developments in Language Theory: 17th International Conference, DLT 2013, Marne-la-Vallée, France, June 18–21, 2013 Proceedings, pp. 118–130. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-38771-5_12
9. Bollig, B., Katoen, J.P., Kern, C., Leucker, M., Neider, D., Piegdon, D.R.: libalf: the automata learning framework. In: CAV'10, pp. 360–364 (2010)
10. Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A.: Model-Based Testing of Reactive Systems: Lecture Notes in Computer Science, vol. 3472. Springer, New York (2005)
11. Burkart, O., Steffen, B.: Model checking for context-free processes. In: Cleaveland, W. (ed.) CONCUR 92. Lecture Notes in Computer Science, vol. 630, pp. 123–137. Springer, Berlin (1992). https://doi.org/10.1007/BFb0084787
12. Burkart, O., Steffen, B.: Composition, decomposition and model checking of pushdown processes. Nordic J. Comput. **2**(2), 89–125 (1995)
13. Cassel, S., Howar, F., Jonsson, B.: RALib: a LearnLib extension for inferring EFSMs. DIFTS **5** (2015)
14. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
15. Drews, S., D'Antoni, L.: Learning symbolic automata. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings, Part I, pp. 173–189. Springer, Berlin (2017). https://doi.org/10.1007/978-3-662-54577-5_10
16. Feng, L., Kwiatkowska, M., Parker, D.: In: Compositional Verification of Probabilistic Systems Using Learning, vol. QEST'10, pp. 133–142. IEEE Computer Society, Washington (2010). https://doi.org/10.1109/QEST.2010.24
17. Frohme, M., Steffen, B.: Active mining of document type definitions. In: Howar, F., Barnat, J. (eds.) 23rd International Conference, FMICS 2018, Maynooth, Ireland, September 3–4, 2018, Proceedings. Springer, Berlin (2018)
18. Godefroid, P., Kiezun, A., Levin, M.Y.: In: Grammar-based whitebox fuzzing, vol. PLDI'08, pp. 206–215. ACM, New York (2008). https://doi.org/10.1145/1375581.1375607
19. Godefroid, P., Levin, M.Y., Molnar, D.A.: In: Automated whitebox fuzz testing. www.isoc.org/isoc/conferences/ndss/08/papers/10_automated_whitebox_fuzz.pdf. San Diego, California, USA (2008). (10th February–13th February 2008)
20. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Series in Computer Science, 2nd edn. Addison-Wesley-Longman, Boston (2001)
21. Howar, F.: Active learning of interface programs. Ph.D. thesis, TU Dortmund University (2012). https://eldorado.tu-dortmund.de/bitstream/2003/29486/1/Dissertation.pdf
22. Howar, F., Steffen, B., Jonsson, B., Cassel, S.: Inferring canonical register automata. In: Koncak, V., Rybalchenko, A. (eds.) Verification, Model Checking, and Abstract Interpretation. Lecture Notes in Computer Science, vol. 7148, pp. 251–266. Springer, Berlin (2012). https://doi.org/10.1007/978-3-642-27940-9_17
23. Howar, F., Steffen, B., Merten, M.: Automata learning with automated alphabet abstraction refinement. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation. Lecture Notes in Computer Science, vol. 6538, pp. 263–277. Springer, Berlin (2011). https://doi.org/10.1007/978-3-642-18275-4_19
24. Hungar, H., Margaria, T., Steffen, B.: Test-based model generation for legacy systems. In: Test Conference, 2003. Proceedings. ITC 2003. International, vol. 1, pp. 971–980 (2003). https://doi.org/10.1109/TEST.2003.1271205
25. Hungar, H., Niese, O., Steffen, B.: Domain-specific optimization in automata learning. In: Hunt, W.A., Jr., Somenzi, F. (eds.) Proceedings 15th International Conference on Computer Aided Verification. Lecture Notes in Computer Science, vol. 2725, pp. 315–327. Springer, Berlin (2003). https://doi.org/10.1007/978-3-540-45069-6_31
26. Isberner, M.: Foundations of active automata learning: an algorithmic perspective. Ph.D. thesis, Technical University Dortmund, Germany (2015). http://hdl.handle.net/2003/34282
27. Isberner, M., Howar, F., Steffen, B.: Inferring automata with state-local alphabet abstractions. In: Brat, G., Rungta, N., Venet A. (eds.) NASA Formal Methods, LNCS, vol. 7871, pp. 124–138 (2013). https://doi.org/10.1007/978-3-642-38088-4_9
28. Isberner, M., Howar, F., Steffen, B.: Learning register automata: from languages to program structures. Mach. Learn. (2013). https://doi.org/10.1007/s10994-013-5419-7
29. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: a redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S. (eds.) Runtime Verification. Lecture Notes in Computer Science, vol. 8734, pp. 307–322. Springer, Berlin (2014). https://doi.org/10.1007/978-3-319-11164-3_26
30. Isberner, M., Howar, F., Steffen, B.: The open-source learnLib: a framework for active automata learning. CAV (2015). https://doi.org/10.1007/978-3-319-21690-4_32
31. Issarny, V., Steffen, B., Jonsson, B., Blair, G.S., Grace, P., Kwiatkowska, M.Z., Calinescu, R., Inverardi, P., Tivoli, M., Bertolino, A., Sabetta, A.: In: ICECCS (ed.) CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems, pp. 154–161. IEEE Computer Society (2009)

32. Kearns, M.J., Vazirani, U.V.: An Introduction to Computational Learning Theory. MIT Press, Cambridge (1994)
33. Kumar, V., Madhusudan, P., Viswanathan, M.: Minimization, learning, and conformance testing of Boolean programs. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006—Concurrency Theory: 17th International Conference, CONCUR 2006, Bonn, Germany, August 27–30, 2006 Proceedings, pp. 203–217. Springer, Berlin (2006). https://doi.org/10.1007/11817949_14
34. Maler, O., Mens, I.E.: Learning regular languages over large alphabets. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5–13, 2014. Proceedings, pp. 485–499. Springer, Berlin (2014). https://doi.org/10.1007/978-3-642-54862-8_41
35. Margaria, T., Niese, O., Raffelt, H., Steffen, B.: Efficient test-based model generation for legacy reactive systems. In: HLDVT'04: Proceedings of the High-Level Design Validation and Test Workshop, 2004. 9th IEEE International, pp. 95–100. IEEE Computer Society, Washington (2004). https://doi.org/10.1109/HLDVT.2004.1431246
36. McNaughton, R.: Parenthesis grammars. J. ACM **14**(3), 490–500 (1967). https://doi.org/10.1145/321406.321411
37. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of unix utilities. Commun. ACM **33**(12), 32–44 (1990). https://doi.org/10.1145/96267.96279
38. Mitchell, J.C.: Concepts in Programming Languages. Cambridge University Press, Cambridge (2002). https://doi.org/10.1017/CBO9780511804175
39. Nerode, A.: Linear automaton transformations. Proc. Am. Math. Soc. **9**(4), 541–544 (1958)
40. Neubauer, J., Windmüller, S., Steffen, B.: Risk-based testing via active continuous quality control. Int. J. Softw. Tools Technol. Transf. **16**(5), 569–591 (2014). https://doi.org/10.1007/s10009-014-0321-6
41. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. J. Autom. Lang. Comb. **7**(2), 225–246 (2001)
42. Plotkin, G.D.: A structural approach to operational semantics. Tech. rep., University of Aarhus (1981). DAIMI FN-19
43. Rensink, A.: The GROOVE simulator: a tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) Applications of Graph Transformations with Industrial Relevance, pp. 479–485. Springer, Berlin (2004)
44. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Inf. Comput. **103**(2), 299–347 (1993). https://doi.org/10.1006/inco.1993.1021
45. Steffen, B., Howar, F., Merten, M.: Introduction to active automata learning from a practical perspective. In: Bernardo, M., Issarny, V. (eds.) Formal Methods for Eternal Networked Software Systems. Lecture Notes in Computer Science, vol. 6659, pp. 256–296. Springer, Berlin (2011). https://doi.org/10.1007/978-3-642-21455-4_8
46. Vardhan, A., Viswanathan, M.: LEVER: a tool for learning based verification. Presented at the (2006)
47. Windmüller, S., Neubauer, J., Steffen, B., Howar, F., Bauer, O.: Active continuous quality control. In: 16th International ACM SIGSOFT Symposium on Component-Based Software Engineering, CBSE'13, pp. 111–120. ACM SIGSOFT, New York (2013). https://doi.org/10.1145/2465449.2465469