Model-Based Quality Assurance of Instrumented Context-Free Systems

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Markus Theo Frohme

Dortmund

2023

Tag der mündlichen Prüfung:
12.09.2023

Dekan:
Prof. Dr.-Ing. Gernot A. Fink

Gutachter:
Prof. Dr. Bernhard Steffen
Prof. Dr. Bengt Jonsson

Abstract

The ever-growing complexity of today's software and hardware systems makes quality assurance (QA) a challenging task. Abstraction is a key technique for dealing with this complexity because it allows one to skip non-essential properties of a system and focus on the important ones. Crucial for the success of this approach is the availability of adequate abstraction models that strike a fine balance between simplicity and expressiveness.

This thesis presents the formalisms of systems of procedural automata (SPAs), systems of behavioral automata (SBAs), and systems of procedural Mealy machines (SPMMs). The three model types describe systems which consist of multiple procedures that can mutually call each other, including recursion. While the individual procedures are described by regular automata and therefore are easy to understand, the aggregation of procedures towards *systems* captures the semantics of context-free systems, offering the expressiveness necessary for representing procedural systems.

A central concept of the proposed model types is an instrumentation that exposes the internal structure of systems by making calls to and returns from procedures observable. This instrumentation allows for a notion of rigorous (de-) composition which enables a translation between local (procedural) views and global (holistic) views on a system. On the basis of this translation, this thesis presents algorithms for the *verification*, *testing*, and *learning* of (instrumented) context-free systems, covering a broad spectrum of practical QA tasks. Starting with SPAs as a "base" formalism for context-free systems, the flexibility of this concept is shown by including features such as prefix-closure (SBAs) and dialog-based transductions (SPMMs).

In a comparison with related formalisms, this thesis shows that the simplicity of the proposed model types not only increases the understandability of models but can also improve the performance of QA tasks. This makes SPAs, SBAs, and SPMMs a powerful tool for tackling the practical challenges of assuring the quality of today's software and hardware systems.

Acknowledgements

First, I would like to thank Bernhard Steffen for his guidance over the last years. My academic career will always be connected with you: From allowing me to join your chair as a student assistant during my bachelor studies, throughout my master studies in which you introduced me to the wonderful field of active automata learning, up until this interesting journey of my PhD. Having spent over a third of my life on your side has definitely left its mark.

What I enjoy and admire most about working with you is your ability to always take a step back from a problem and tackle it from a completely different point of view. Sometimes, your uncompromising approach of throwing things away and starting completely new from scratch was a little bit difficult and definitely caused some headaches in the past. However, it was always justified in the end by finding a better and more elegant solution.

I would like to thank Bengt Jonsson for agreeing to act as a referee for this thesis on a rather short notice. While we have not had much contact prior to this thesis, especially your work on register automata follows a similar line of thought of enriching models to capture more and more practically relevant traits of systems. It is an interesting question for the future, whether there exist some fruitful connection points between the procedural hierarchy discussed in this thesis and the data registers of your work.

I would like to thank Falk Howar for his support and insights as a mentor during my PhD thesis. Being a fellow contributor and co-maintainer of LearnLib, you always had an open ear for discussions on improving the library and continued to push the project with your own ideas, making LearnLib the tool it is today.

Furthermore, I would like to thank my colleagues at the chair of programming systems at TU Dortmund university and especially my roommates over the years: Johannes Neubauer, Hendrik Grewe, and Alexander Bainczyk (chronological order). You were always on board for cracking even the most un-appropriate jokes which helped me to survive long, draining days. Yet, at the same time, we could always have fruitful discussions to push forward the projects that we were working on.

Last but not least, I would like to thank my family for supporting me throughout my whole life. From making it possible for me to attend university in the first place to allowing me to fully focus on my studies when necessary. I hope to make you proud of me.

Contents

CHAPTER 1

Introduction

Over the last decades, both software and hardware have found their way into our lives to a point where today one can barely think of a world without them: medical support systems, global e-commerce or the private entertainment sector are just a few examples where computer systems and programs play an essential role. A catalyst for this rise in digitization was (and still is) the growing complexity of software and hardware, which allows software programs and hardware devices to perform more and more challenging tasks.

Crucial for establishing and sustaining today's level of integration and dependency on both software and hardware is having thorough quality assurance (QA) to guarantee that both software and hardware operate as intended. On the one hand, there are *requirements* which specify the intended behavior of an application. On the other hand, there are (software or hardware) *systems* that have to correctly implement these requirements.

For QA, the huge complexity of powerful systems comes at a cost: For example, consider a world-wide operating, distributed web application. Verifying that after clicking a button, the browser sends the correct data over a network socket, which is then routed to a specific server so that it is successfully stored in a sector of the server's hard-drive while thousands of other users simultaneously use the website is not practical. However, by not properly addressing the technical properties of systems, correctness may not be assured thoroughly and potentially result in catastrophic (both monetary and life-threatening) failures. Some of the classic examples of such failures include the floating-point division bug in early Pentium® central processing units or the failed launch of the Ariane 5 heavy-lift launch vehicle. But even in recent history, after years of experience and development, such failures continue to occur [79, 106, 141].

The challenge for QA is to find a trade-off between thoroughness and performance/feasibility, which has attracted the interest and investment of computer scientists and software engineers alike. A very powerful and promising means to address this issue is the introduction of *models*. The key idea is to introduce an additional (often formal) layer, a model, to serve as a mediator between the requirements of a system and the system itself. Towards the system, it is able to provide an abstraction that blends out certain technical aspects of the system and focus on behavioral aspects that are important for its requirements, therefore providing performance/feasibility. Towards the requirements, it is able to provide a formal view on the system that allows for mathematical proofs of properties, therefore providing thoroughness. Throughout this thesis, this concept is referred to as model-based quality assurance (MBQA).

Figure 1.1
The three basic components of MBQA and possible interactions between them.

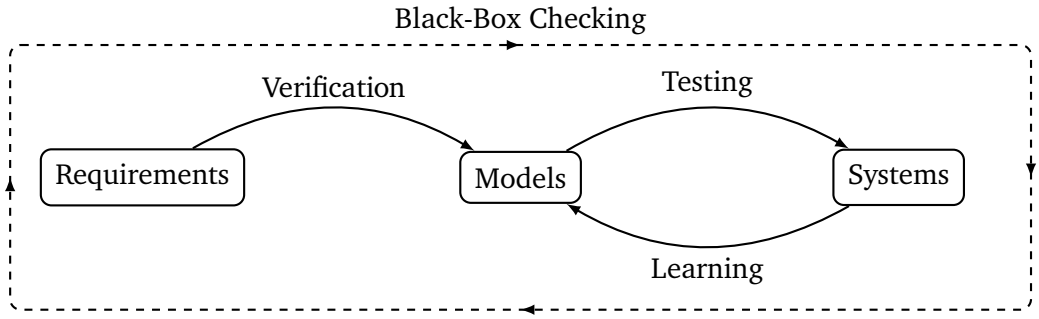


Figure 1.1 sketches the three main components of MBQA and some of the potential interactions between them. At the center of MBQA, there are *models*. For software and hardware systems, one typically chooses some form of *transition system*, i.e., a graph-like structure with system states and transitions between them. These model types often naturally mimic the internal structure of a system and are able to closely capture its operational semantics. Examples for systems that focus on *actions*, i.e., when external inputs progress the system state, are deterministic finite acceptors (DFAs), Mealy machines [121] or labeled transition systems (LTSs). Examples for systems that primarily expose observable behavior in the form of state propositions are Kripke transition systems (KTSs) [108]. Possible extensions of these model types, e.g., to include additional properties, or combinations thereof are also possible.

Introducing a formal model as a mediator between the requirements and the system directly impacts these two components as well. Specifications (of requirements) now have to target the model instead of the actual implementation. This has the potential to allow for more idiomatic specifications because it is now possible to abstract from technical details and articulate requirements in some form of domain-specific language, depending on the chosen model type. In case of state-based and transition-based models (and systems) this often involves logics whose formulae describe (properties of) paths in a transition system. Popular examples of these kinds of logics are linear temporal logic (LTL) [140], computational tree logic (CTL) [46] or the (modal) μ-calculus [107].

Systems now need some form of additional interface in order to translate between abstract model actions (or propositions) and concrete system actions (or propositions). Here, it is important to establish a consistent set of actions across all three components of the MBQA setup. For example, if the requirements state that an action a must be possible at a certain point, the model and ultimately the system must be able to recognize and interpret this very action correctly to be able to reliably verify the requirement. At the same time, this modeling of actions offers an additional parameter to control the level of abstraction, e.g., using a high-level interaction such as “login to the application” versus using a low-level interaction such as “write a value to a register”.

Compared to classic QA, the process of assuring quality in case of MBQA is a two-step process. The first step — from requirements to models — is called *model verification* or *model checking* [18, 47] and describes the process of verifying that the model adheres to the behavior that is specified by the requirements. As discussed above, the formalism for the requirements often describes desired (properties of) paths of the model. In this case, the verification question can be answered by checking whether the requirements specify paths that are missing in the model or if the model exhibits paths that are not allowed by the requirements. There exists a long history of model checking tools that answer this question for various requirement logics and model types [19, 20, 27, 28, 45, 78, 81, 100, 154, 162].

The second step — from models to systems — is called *model-based testing* [36, 111] and often concerns the (automated) generation of test cases in order to check properties of the system. In the context of MBQA, the sub-field of *conformance testing* [68, 111] is often of most interest, as it deals with the question of equivalence between the model and the system. In an offline (or development) environment, this involves generating test cases that traverse characteristic transitions of the model and when executed on the system are used to compare the observed behavior with the expected behavior of the model. In reminiscence of Dijkstra[1], these are often best-effort heuristics as their (provable) correctness often depends on additional knowledge about the system. In an online (or production) environment, concepts such as *monitoring* [54] can be used to observe the behavior of the system while it is in use and verify the observed behavior against the expected behavior of the model.

However, employing a model also adds new challenges to the QA process of a system. Creating a formal specification and a formal model may be a very tedious and error-prone task. First, formalizing correct requirements needs particular knowledge about the used specification language [147, 170, 174] and may require a substantial amount of additional person-hours [69, 175]. Using and overly complex specification formalism may cause additional confusion [110, 152]. Second, these requirements need to be implemented in a formal model *in addition* to the actual system implementation, which also requires knowledge about the specific modeling language to do so efficiently [120]. Any error in the specification or model could potentially invalidate any results obtained from MBQA. This problem gets amplified if one chooses to employ MBQA during the development process of a system as multiple iterations of the above steps become necessary. Here, the trade-off between return and investment may scare off potential users.

Specifically for the problem of constructing a model, a potential solution would be to create models from a system *automatically*. For software systems, there are a number of static code analysis tools available [14, 27, 142] that allow one to analyze and transform source code into formal models such as data-flow graphs or control-flow graphs which can then be used to verify properties. Some of the aforementioned model checkers [19, 27, 28, 78] also come with integrated analysis tools and operate directly on the source code of a program.

[1] "Program testing can be used to show the presence of bugs, but never to show their absence!" — Dijkstra [50].

However, for a truthful model, these methods require full access to the system's source code. Nowadays, software is complex and consequently often composed of many individual components, potentially third-party libraries or external services to which access to the corresponding source code is not obtainable. This can often render the above approaches obsolete since fundamental parts of the system may not be properly analyzable. A similar situation occurs for hardware systems, where, for example, circuit plans can be abstracted to models but third-party components confront the hardware designers with the same problems. Another challenge, specifically for source-to-model transformers, is that these tools create very verbose models when operating on large code bases. It often requires manual intervention by introducing custom symbolic abstractions in order to allow for humane specifications of requirements.

A powerful means to tackle the above challenges is active automata learning (AAL). In AAL, a learning algorithm (or simply *learner*) interacts with a system under learning (SUL) by means of testing, i.e., by interacting with the system based on previously defined symbolic actions. By observing the responses of the SUL to the tests, the learner constructs a formal (automaton-based or transition system-based) hypothesis model of the SUL that describes the observed behavior. AAL provides solutions for the above challenges by

1. operating on a previously defined set of symbols, meaning that the inferred model has the exact granularity that the user has specified, and by
2. automatically inferring a model based on the observable behavior of the system without needing access to its internals.

In its seminal introduction [15], AAL is motivated from a theoretical point of view as an efficient solution to the broader problem of grammatical inference of unknown formal languages. However, AAL quickly gained traction in practical scenarios because formal languages can be easily associated with successful executions of software or hardware systems. The requirement of AAL to actively interact with an SUL can be easily implemented by testing, i.e., executing the respective tests on the given (software or hardware) system. Although AAL is in general neither correct nor complete, there are several success stories [1, 2, 6, 26, 41, 57, 91, 99, 131, 136, 143, 158, 167, 172] which position AAL as a powerful provider of models to enable MBQA in practice.

As indicated in Figure 1.1, the three disciplines of verification, testing, and learning can also be combined into a joint approach called black-box checking (BBC) [135]. BBC establishes the concept of a feedback loop in which learning first constructs a hypothesis model of a system which is then directly verified by a model checker. This may potentially detect violations against the requirements which are then checked on the system in order to determine whether the system actually fails the requirement, or the hypothesis model of the learner needs a refinement which starts a new learning cycle. Here, the three disciplines are not treated in isolation but in unison which has the potential to further boost the quality and performance of the MBQA process.

1.1 Scope of this Thesis

The appeal of MBQA is one's freedom to decide which model type to use, giving one the ability to find a favorable trade-off between thoroughness and performance/feasibility of QA. Crucial for the success of MBQA-based techniques is the availability of intuitive yet powerful and expressive models that support the previously discussed workflows.

1.1.1 Contributions

This thesis presents the model types of systems of procedural automata (SPAs), systems of behavioral automata (SBAs), and systems of procedural Mealy machines (SPMMs), which describe procedural systems modeled after context-free grammars (CFGs) or context-free languages (CFLs), respectively. While SPAs constitute a "base" formalism for describing procedural systems holistically, SBAs introduce the notion of prefix-closure and SPMMs introduce the notion of transduction. As a result, the three formalisms provide abstractions for procedural systems that are tailored towards different use-cases that can be found in real-world scenarios.

Essential to the concepts of the three model types is an instrumentation that makes calls to and returns from procedures observable, exposing the internal structure of systems. This instrumentation allows for a notion of rigorous (de-) composition of systems into their individual procedures, providing expressive, intuitive, and performant models for the three discussed disciplines of MBQA.

The contributions of this thesis are based on five peer-reviewed research papers whose results are summarized, aligned with each other, and expanded on in this document.

Paper 1
Markus Frohme and Bernhard Steffen. "Compositional learning of mutually recursive procedural systems". In: International Journal on Software Tools for Technology Transfer *23.4 (2021), pp. 521–543.* DOI: 10.1007/s10009-021-00634-y.

The paper presents the notion of SPAs, an AAL algorithm for inferring SPA models, and a performance comparison with the competing formalism of visibly push-down automata (VPAs). The concept of SPAs is based on Bernhard Steffen's earlier work on context-free process systems (CFPSs) [37]. Regarding the learning process, the notion of query expansion and counterexample projection was proposed by Bernhard Steffen, whereas I incorporated the concept of incremental alphabet extensions in order to successively obtain access sequences, terminating sequences, and return sequences that are required for the query expansion throughout the learning process. Regarding counterexample analysis, the idea of the alpha-gamma transformation was proposed by Bernhard Steffen, whereas I established the property of monotonicity to allow for a Rivest & Schapire-style counterexample analysis process. Furthermore, I implemented the code that was made publicly available and conducted the experiments.

In this thesis, I further integrate SPAs into the discussed MBQA processes by formalizing (and implementing) the processes of SPA verification and SPA conformance testing. I show

the equivalence between SPA languages and the general context-free interpretation of instrumented systems, which underlines the soundness of SPAs to capture the semantics of context-free systems. Regarding comparability, I further present transformations from SPAs into VPAs and vice versa. On the qualitative side, this highlights differences and similarities between the two formalisms and makes QA techniques for one model type applicable to the other model type. On the quantitative side, this allows me to analyze fundamental model properties in this thesis which explain the results observed in the paper.

Paper 2
Markus Frohme and Bernhard Steffen. "From Languages to Behaviors and Back". In: Lecture Notes in Computer Science 13560 (2022). Ed. by Nils Jansen, Mariëlle Stoelinga, and Petra van den Bos, pp. 180–200. DOI: 10.1007/978-3-031-15629-8_11.

The paper presents the notion of SBAs, an AAL algorithm for inferring SBA models, and compares the performance of an "SBA inference + reduction" process with the standard SPA inference process. The paper was motivated by my idea to extend SPAs to support a transduction mechanism similar to the results of [133]. However, concerned with the length of the paper, I pursued the idea of prefix-closed formal languages as the foundation for the concerned transductions instead because it allowed me to re-use existing notation and results of SPAs. It was Bernhard Steffen's idea to include the return symbol in the procedural alphabet in order to distinguish between returnability and reachability and it was his idea to provide an alternative (graph-based) characterization of SBAs to better address the reactive systems community. In mutual discussions, we developed the idea of using alphabet extensions to tackle divergent states. Furthermore, I formalized and proved several of the required properties for the query expansion, counterexample projection, and the learning algorithm of SBAs and implemented the code for executing the benchmarks.

In this thesis, I further integrate SBAs into the discussed MBQA processes by formalizing (and implementing) the processes of SBA verification and SBA conformance testing. Furthermore, I pursue my original idea of the paper by formalizing SPMMs as an SBA-based specialization for (instrumented) context-free transductions and discussing the necessary adjustments for the verification, testing, and learning thereof, providing a third model type for practical MBQA. In a distinct benchmark, I compare the native SPMM formalism with its SBA-based characterization to showcase its performance benefit.

Paper 3
Markus Frohme and Bernhard Steffen. "Never-Stop Context-Free Learning". In: Lecture Notes in Computer Science 13030 (2021). Ed. by Ernst-Rüdiger Olderog, Bernhard Steffen, and Wang Yi, pp. 164–185. DOI: 10.1007/978-3-030-91384-7_9.

The paper evaluates the performance of the SPA learning algorithm of Paper 1 in the context "never-stop learning". The idea of never-stop learning has been conceptualized by Bertolino et al. [26] and has been pushed by Bernhard Steffen in other contexts such as the TTT algorithm [96]. Therefore, the initial idea of the paper was inspired by Bernhard Steffen

and the presented benchmarks have been selected in mutual discussions with me. The implementation, conduction, and evaluation of the benchmarks was done by me.

Some of the rather intriguing concepts are only sketched in the paper. In particular, I formalize my proposed optimizations that allow for the good performance of the SPA learner in the "external redundancy" setting in this thesis and provide a more fine-grained evaluation of their impact. Furthermore, the concept of "procedural characterizing sets" is subsumed in this thesis by my work on the conformance testing of SPAs.

Paper 4
Markus Frohme and Bernhard Steffen. "A Context-Free Symbiosis of Runtime Verification and Automata Learning". In: Lecture Notes in Computer Science 13065 (2021). Ed. by Ezio Bartocci, Yliès Falcone, and Martin Leucker, pp. 159–181. DOI: 10.1007/978-3-030-87348-6_10.

The paper is an extension of Paper 3 that specifically tackles the search of counterexamples in the context of SPA learning. Therefore, it was inspired by the initial never-stop learning vision of Bernhard Steffen. The idea of the self-optimizing stack structure was proposed by Bernhard Steffen and the formalization of an SPA monitor via structural operational semantics (SOS) was developed in mutual discussions with me. Furthermore, I implemented, conducted and evaluated the various benchmarks scenarios.

In this thesis I provide a generalized notion of monitors that allows one to incorporate both SPA and SBA semantics. This direct comparison allows one to easily see how, e.g., SBAs, due to their prefix-closure, allow for much more intuitive and performant monitors.

Paper 5
Markus Frohme and Bernhard Steffen. "Active Mining of Document Type Definitions". In: Formal Methods for Industrial Critical Systems - 23rd International Conference, FMICS 2018, Maynooth, Ireland, September 3-4, 2018, Proceedings. *Ed. by Falk Howar and Jiri Barnat. Vol. 11119. Lecture Notes in Computer Science. Springer, 2018, pp. 147–161. DOI: 10.1007/978-3-030-00244-2_10.*

The paper presents a (fictional) case-study for the practical application of SPAs as a model type. The idea of interpreting opening tags and closing tags of extensible markup language (XML) documents and establishing a relation between SPAs and document type definitions (DTDs) was motivated by me, after having seen similar approaches for the related formalism of VPAs. The use-case of the general data protection regulation (GDPR) as well as the particular example presented in the paper was developed in mutual discussions with Bernhard Steffen. The elaboration of the running example was done by me.

In this thesis, I expand on the practical application of SPAs, SBAs, and SPMMs by discussing the technical aspects of system instrumentation and elaborating other application domains where the three proposed model types naturally fit into.

Overall, the published papers and their consolidations and extensions in this thesis aim at providing expressive, performant, but also intuitive formalisms that offer powerful tools

for tackling the practical challenges of MBQA. This also involves actual implementations of theoretic concepts. Unfortunately, throughout my research for this thesis, I have come across several open-source tools in this field that have been abandoned. Either the developers or researchers no longer respond to e-mails or tools are no longer able to run in current environments. While there is certainly money to be made with MBQA, as several closed-source or paid tools show, I think open-source is an integral part for the progress of research, either by verifying the work of each other or allowing for fruitful collaborations. In order to give back to this eco-system, I have implemented several of the presented algorithms in this thesis and in part already have submitted or plan to submit them to the open-source tools LearnLib and AutomataLib [95]. This work also includes coordinating third-party submissions such as the model checker of [162] which has been contributed by the authors and is used for the verification of SPAs, SBAs, and SPMMs in this thesis. LearnLib and AutomataLib are (mainly) developed at the faculty of computer science at TU Dortmund university and look back on a history of over fifteen years with hopefully many more to come. By (re-) integrating my work into these libraries, I hope to enable future researchers and practitioners to profit off of the results of this thesis.

Note that throughout this thesis, I occasionally use the pronoun “we”. It is meant to include the reader in discussions on the same level. Especially, it does not represent “we” as in multiple authors. This thesis is the sole work of my own.

1.1.2 Limitations

This thesis makes certain assumptions that are not investigated further and are postponed to potential future work.

As previously mentioned, the definition of abstract symbolic interactions is an important part of MBQA as it is the unifying component across requirements, models, and systems. At the same time, it is also a highly individual problem because the interactions are often specific to the actual system at hand. Sometimes, it is even necessary to handle an infinite domain of symbols, e.g., when working with numerical values such as the natural numbers $\mathbb{N}$. It is hard to make general assumptions about the symbolic interactions and elaborating on possible cases goes beyond the scope of this thesis. Therefore, this thesis follows an approach that is common in literature by assuming that the symbolic definitions are already present. This also holds for the instrumentation. While this thesis briefly discusses the technical aspects of instrumenting systems and presents scenarios where such an instrumentation is inherently provided by systems, it is generally assumed that the considered systems are able to correctly interpret SPA-based, SBA-based, and SPMM-based interactions as it makes highlighting the conceptual and algorithmic properties of these formalisms easier and clearer.

The comparison of performance between SPAs and VPAs is restricted to the VPA-based formalisms available in LearnLib and AutomataLib [95], namely 1-single-entry visibly push-down automata (SEVPAs), as it allows for a re-use of existing implementations for the evaluation. In the comparison, the variation of n-SEVPAs shows some interesting properties that may be investigated in some distinct, future work. While the presented evaluation

is sufficient for providing a general intuition and bringing across the central points, a thorough analysis requires additional work. However, formalizing and implementing the verification, testing, and learning of n-SEVPAs is beyond the scope of this thesis.

The work on context-free transductions is restricted to deterministic transductions that follow an incremental lock-step pattern. This means, for each input symbol sent to the system, the system responds with a single deterministic output symbol before the next input symbol is processed. A well-known formalism to implement these semantics for regular formal languages are Mealy machines [121]. While there exist more general concepts of context-free transductions such as *rational transductions* or *sequential transductions* [24], this is beyond the scope of this thesis. Especially in the context of (regular) MBQA, Mealy machines and their semantics have proven themselves as a solid formalism with a lot of applicable use-cases. Since one of the main themes of this thesis is to lift these concepts to the context-free level by means of composition, work on more general formalisms is left for future research.

Finally, the topics of verification, testing, and learning are presented as the ingredients of BBC. This thesis briefly discusses the interactions of the respective disciplines and discusses some particularly fruitful applications in the context of SPAs, SBAs, and SPMMs. However, a thorough analysis of this process and the potential impact on the individual techniques is beyond the scope of this thesis and deferred to future research.

1.2 Overview

The remainder of this thesis is structured as follows:

Chapter 2 introduces preliminary notation and sketches the concepts of verification, testing, and learning required for understanding the corresponding work on SPAs, SBAs, and SPMMs.

Chapter 3 introduces the formalisms of SPAs, SBAs, and SPMMs. It establishes central properties of the formalisms and discusses specific application profiles such as monitoring.

Chapter 4 presents the translations of SPAs, SBAs, and SPMMs to CFPSs. It establishes the notion of word-to-path equivalence such that model checkers for CFPSs can be used to verify SPAs, SBAs, and SPMMs languages (transductions).

Chapter 5 presents a methodology for constructing conformance tests for SPAs, SBAs, and SPMMs.

Chapter 6 presents AAL algorithms for SPAs, SBAs, and SPMMs within the minimally adequate teacher (MAT) framework and analyzes their query complexity. Furthermore, it presents several heuristics for improving the practical performance of the learners.

Chapter 7 discusses the formalism of VPAs and presents language-equivalent transformations from SPAs into SEVPAs and vice versa.

Chapter 8 presents work from related fields of research, including formalisms for procedural and recursive systems and the verification, testing, and learning thereof.

Chapter 9 discusses aspects of the practical application of SPAs, SBAs, and SPMMs. This covers the technical aspects of instrumentation, particularly fruitful application domains, as well as potential symbioses between the difference disciplines of MBQA.

Chapter 10 discusses qualitative and quantitative properties of SPAs, SBAs and SPMM, including comparisons with the competing formalism of VPAs.

Chapter 11 concludes this thesis by summarizing its results and presenting an outlook on future research.

CHAPTER 2

Preliminaries

This chapter introduces several formal notations that are used throughout this thesis and sketches the topics of verification, testing, and learning. The chapter is meant to give an intuition for these processes in preparation for the work on the verification, testing, and learning of (instrumented) context-free systems. Related work on these topics is presented in Chapter 8.

2.1 Formal Languages

The foundational concept for the work of this thesis are *formal languages*. Essential to formal languages are *alphabets* which represent collections of alphabet *symbols*.

Definition 1 (Alphabet)
An alphabet *I is a non-empty, finite set of symbols over some domain. The elements of I are called* alphabet symbols*, denoted $a \in I$.*

As discussed in Chapter 1, alphabets are a core concept of model-based quality assurance (MBQA) as they describe the abstract interactions that are shared across requirements, models, and systems. Depending on the context, alphabets (or symbols) may be classified into sub-categories such as *input alphabets* which represent (input) interactions with a system or *output alphabets* which represent observable responses from a system. As discussed in Section 1.1.2, this thesis assumes that the alphabet definitions are always clear from the context. Given an alphabet, we continue with the definition of *words* over an alphabet.

Definition 2 (Words over an alphabet)
Let I denote an alphabet. A word *w over I is a finite concatenation of alphabet symbols, i.e., $w = a_1 \cdot a_2 \cdot \ldots \cdot a_n$ with $a_i \in I$ for all $i \in \{1, \ldots, n\}$.*

- *We write a^n to denote the n-fold repetition of a, i.e., $a^n = \overbrace{a \cdot \ldots \cdot a}^{n \text{ times}}$.*
- *We write ε to denote the* empty word *that contains no symbols.*
- *We write I^n to denote the set of all words over I with length n, i.e., $I^n = \{a_1 \cdot \ldots \cdot a_n \mid a_i \in I\}$.*
- *We write I^* to denote the set of all finite words over I, i.e., $I^* = \bigcup_{n \in \mathbb{N}} I^n$.*
- *We write I^+ to denote the set of all non-empty finite words over I, i.e., $I^+ = I^* \setminus \{\varepsilon\}$.*
- *We write $|I|$ to denote the size of I, i.e., the cardinality of the set.*

In the following, let $w \in I^*$ denote a word over I with $w = a_1 \cdot a_2 \cdot \ldots \cdot a_n$.

- *We write $w[i]$ to denote the i-th symbol of w, i.e., $w[i] = a_i$ for all $i \in \{1, \ldots, n\}$.*
- *We write $w[i, j]$ to denote the sub-word of w that starts at index i and ends at index j, i.e., $w[i, j] = a_i \cdot \ldots \cdot a_j$. For $i > j$, we have $w[i, j] = \varepsilon$.*
- *We write $w[, i]$ to denote the prefix ending at index i, i.e., $w[, i] = a_1 \cdot \ldots \cdot a_i$. For $i < 1$, we have $w[, i] = \varepsilon$.*
- *We write $w[i,]$ to denote the suffix starting at index i, i.e., $w[i,] = a_i \cdot \ldots \cdot a_n$. For $i > n$, we have $w[i,] = \varepsilon$.*
- *We write $|w|$ to denote the length of w, i.e., $|w| = n$.*

In the following, let $v \in I^*$ denote another word over I with $v = b_1 \cdot b_2 \cdot \ldots \cdot b_m$.

- *We write $w = v$ to denote that w and v are equal. Two words w and v are equal iff $|w| = |v|$ and $a_i = b_i$ for all $i \in \{1, \ldots, n\}$.*
- *We write $w \cdot v$ to denote the concatenation of words, i.e., $w \cdot v = a_1 \cdot \ldots \cdot a_n \cdot b_1 \cdot \ldots \cdot b_m$.*
- *We call v a* prefix *of w iff there exists an index $i \in \{1, \ldots, n\}$ such that $v = w[, i]$. Furthermore, ε always constitutes a valid prefix of any word.*
- *We call v a* suffix *of w iff there exists an index $i \in \{1, \ldots, n\}$ such that $v = w[i,]$. Furthermore, ε always constitutes a valid suffix of any word.*
- *We write $Pref(w)$ to denote the set of all prefixes of w.*
- *We write $Suff(w)$ to denote the set of all suffixes of w.*

Words over an alphabet represent sequences of individual alphabet symbols. In the context of MBQA, alphabet symbols are meant to represent interactions with a software system or a hardware system. This means that a word corresponds to a *run* or an *execution* of a system. The aggregation of several, e.g., successful, runs of a system, serves as a means to describe the *behavior* of a system. On a formal level, this is concept is denoted via *formal languages*.

Definition 3 (Formal language)
Let I be an alphabet. A formal language *L is a set of words over I, i.e., $L \subseteq I^*$.*
Let L_1, L_2 denote two formal languages.

- *We write $L_1 \cdot L_2$ to denote the language of pair-wise concatenations, i.e., $L_1 \cdot L_2 = \{l_1 \cdot l_2 \mid l_1 \in L_1, l_2 \in L_2\}$.*

With formal languages, there exists an intuitive yet formally precise formalism for describing system behavior. One can decide if a system meets its requirements by checking whether a system language contains “bad” words or is missing “good” words. The question of conformance then corresponds to the question of language equivalence between, e.g., the requirements and the model or the model and the system. Crucial for the ability of a formal language to capture the behavior of a system is the expressiveness of the language, as the following example illustrates.

Example 1 (Expressiveness of two languages)
Let $I = \{a, b\}$ be an alphabet and $L_1 = \{a^n \cdot b^m \mid n, m \in \mathbb{N}\}$ and $L_2 = \{a^n \cdot b^n \mid n \in \mathbb{N}\}$ be two languages over I. Intuitively, L_1 is simpler than L_2 because each word of L_1 consists of an arbitrary number of as followed by an arbitrary number of bs. In contrast, a system described by L_2 requires some kind of counting mechanism to ensure the same number of as and bs.

A popular classification of formal languages is given by Chomsky [43] who proposes a hierarchy of four language *classes*, i.e., sets of languages: regular languages ($\mathcal{L}_{reg}$), context-free languages ($\mathcal{L}_{cf}$), context-sensitive languages ($\mathcal{L}_{cs}$), and recursively enumerable languages ($\mathcal{L}_{re}$). Each language class is strictly less expressive than the subsequent one, i.e., $\mathcal{L}_{reg} \subsetneq \mathcal{L}_{cf} \subsetneq \mathcal{L}_{cs} \subsetneq \mathcal{L}_{re}$. For example, the language L_1 of Example 1 is a regular language, whereas the language L_2 of Example 1 is a context-free language but not a regular language.

As mentioned in Chapter 1, the challenge but also the opportunity of models is to find a favorable trade-off between complexity and expressiveness. In Chomsky's hierarchy, regular languages are the simplest language class but they exhibit a lot of useful closure-properties. Not only do operations such as union, intersection, and concatenation yield a regular language again, but also these operations can be computed in polynomial time (with respect to the size of their formalism, see below). This makes (and has made) regular languages a convenient formalism for MBQA, as it allows one to transfer concepts such as composition from the system level to the model level.

At the same time, regular languages lack natural support for concepts such as recursion and one either needs to introduce additional abstraction to describe these concepts or move up in the hierarchy. However, with increasing expressiveness, the language properties become much harder to manage. For example, already with context-free languages, the intersection of two context-free languages may no longer be context-free, i.e., they cannot be described by a context-free formalism anymore. Furthermore, problems such as the language equivalence problem of two context-free languages is no longer decidable. Here, a lot of the comfort known from regular languages is no longer available. For a more thorough analysis see, e.g., [83].

Previewing the contents of Chapter 3, systems of procedural automata (SPAs) (and by extension systems of behavioral automata (SBAs) and systems of procedural Mealy machines (SPMMs)) describe context-free systems in which the entry points and exit points of procedures are made explicitly observable. This enables a notion of rigorous (de-) composition of systems into their procedural components, which offers the expressiveness of context-free systems combined with the ease of regular systems. We continue with looking at regular languages and context-free languages in more detail and discuss extensions such as the incorporation of outputs to describe *dialog* systems in which users interact with a system in a bi-directional manner.

2.1.1 Generators

A generator is a form of recipe for constructing words of a language. As languages are used to describe (successful) runs of a system, a generator-based interpretation of a language

can be seen as a roadmap for possible executions. A versatile formalism for defining (up to recursively enumerable) languages is that of *unrestricted formal grammars* [83].

Definition 4 (Unrestricted formal grammar)
An unrestricted formal grammar *is a tuple* $G = \langle N, T, P, S \rangle$ *where*

- N *denotes the non-empty, finite alphabet of non-terminal symbols,*
- T *denotes the non-empty, finite alphabet of terminal symbols,*
- $P \subseteq (N \cup T)^+ \times (N \cup T)^*$ *denotes the non-empty, finite set of production rules, and*
- $S \in N$ *denotes the initial non-terminal symbol.*

We define the derivation relation $\longrightarrow \subseteq (N \cup T)^* \times (N \cup T)^*$ *such that* $t_1 \longrightarrow t_2$ *iff* $\langle p_1, p_2 \rangle \in P$ *and* t_2 *is constructed from the syntactical substitution of* p_1 *with* p_2 *in* t_1*. We define the transitive closure of* $\longrightarrow$ *as* $\Longrightarrow$*. We define the* language of a grammar G *as the union of all terminal derivations, i.e.,*

$$L(G) = \{w \in T^* \mid S \Longrightarrow w\}.$$

The expressiveness of a formal grammar is inherently defined by its production rules. By limiting the structure of production rules, it is possible to restrict the language of a formal grammar to different language classes. As mentioned before, regular languages and context-free languages are of particular interest for this thesis and therefore, we continue with the specific grammars for these languages classes.

Definition 5 (Regular grammar)
A formal grammar $G = \langle N, T, P, S \rangle$ *is called* regular *iff its production rules are limited to either*

- $P \subseteq N \times (T \cup (N \cdot T) \cup \{\varepsilon\})$ *(left-recursive) or*
- $P \subseteq N \times (T \cup (T \cdot N) \cup \{\varepsilon\})$ *(right-recursive).*

Definition 6 (Context-free grammar)
A formal grammar $G = \langle N, T, P, S \rangle$ *is called* context-free *iff its production rules are limited to*

- $P \subseteq N \times (N \cup T)^*$.

Furthermore, this thesis only considers *minimal* grammars, i.e., grammars that do not contain any redundant non-terminals.

Definition 7 (Minimal formal grammar)
Let $G = \langle N, T, P, S \rangle$ *be a formal grammar. We call* G minimal *iff*

$$\forall n \in N : \exists w \in L(G), w' \in (N \cup T)^*, i \in \{1, \ldots, |w'|\} : \\ S \longrightarrow \ldots \longrightarrow w' \longrightarrow \ldots \longrightarrow w \wedge w'[i] = n,$$

i.e., every non-terminal symbol occurs in at least one intermediate representation of a terminal derivation.

It is easy to see how a (regular or context-free) formal grammar can be minimized by removing any unreachable non-terminal symbols and their corresponding production rules without affecting the generated language.

2.1.2 Acceptors

Besides the generator-based characterization of formal languages, there also exists a parser-based or acceptor-based characterization. Here, rather than actively constructing words of a formal language, a formalism reads arbitrary words and decides whether they belong to a formal language afterwards. The two concepts are closely linked with each other as each approach can be implemented via the other: Given a generator, it can *accept* an arbitrary word iff it is a member of the set of generated words. Given a parser, it can iterate over all possible words over an alphabet and *generate* a word whenever it is accepted by the parser. However, depending on the context, one of the two approaches may be more convenient to use and therefore it is helpful to briefly present this second characterization as well.

Regarding parsers or acceptors, this thesis only needs to consider a formalism for regular languages. Here, a broadly used formalism is that of deterministic finite acceptors (DFAs).

Definition 8 (DFA)
A DFA is a tuple $A = \langle Q, q_0, I, Q_F, \delta \rangle$ *where*

- Q *denotes the non-empty, finite set of states,*
- $q_0 \in Q$ *denotes the initial state,*
- I *denotes the input alphabet,*
- $Q_F \subseteq Q$ *denotes the set of accepting states, and*
- $\delta : (Q \times I) \to Q$ *denotes the state transition function.*

We define the generalized transition functions $\delta : (Q \times I^*) \to Q$ *and* $\delta : I^* \to Q$ *as*

$$\begin{aligned} \delta(q, u \cdot v) &= \delta(\delta(q, u), v) && \forall q \in Q, u \in I, v \in I^*, \\ \delta(q, \varepsilon) &= q && \forall q \in Q, \\ \delta(w) &= \delta(q_0, w) && \forall w \in I^*. \end{aligned}$$

Alternatively, the transition function may also be represented as a relation $\delta \in Q \times I \times Q$ *and we use both interpretations interchangeably, depending on which is more convenient.*

From a parser-based perspective, a DFA receives a word and consumes it symbol-wise from left to right. Depending on whether this process terminates in an accepting state, the DFA either "accepts" or "rejects" the word. For DFAs, we establish a series of properties.

Definition 9 (Properties of DFAs)
Let $A = \langle Q, q_0, I, Q_F, \delta \rangle, A_1, A_2$ *denote some DFAs over an alphabet* I.

- *We define the* size $|A|$ *of a DFA* A *as the number of its states, i.e.,* $|A| = |Q|$.
- *We define the* language $L(A)$ *of a DFA* A *as* $L(A) = \{w \in I^* \mid \delta(w) \in Q_F\}$.
- *A direct consequence of the above language definition is that for every* $w \in L(A)$ *with* $w = a_1 \cdot \ldots \cdot a_n$ *there exists a path* $q_0 \cdot \ldots \cdot q_n$ *in* A *with* $\delta(q_i, a_{i+1}) = q_{i+1} \forall q_i \in Q, i \in \{0, \ldots, n-1\}$.
- *We call a DFA* A_1 equivalent *to another DFA* A_2 *(denoted as* $A_1 \equiv_{DFA} A_2$*) iff* $L(A_1) = L(A_2)$.
- *Unless specified otherwise, we consider* total DFAs *in this thesis, i.e., DFAs where the*

transition function is defined for all possible input arguments. In a partial DFA*, δ may be undefined for certain input arguments. In this case, the DFA may no longer be able to determine a successor state and the parsed input sequence is rejected. Note that a partial DFA can be easily made total by adding a (rejecting) sink-state and setting this sink state as the successor of every undefined transition, including self-loops for the sink state itself. As a result, we omit any specialized syntax for dealing with partial DFAs.*

- *We call a DFA A_1* minimal *iff there exists no other, equivalent DFA A_2 such that $|A_2| < |A_1|$. Sometimes, minimal DFAs are also called* canonical *because a minimal DFA is also a unique (up to isomorphism) representation for a regular language. Minimality (or canonicity) is no practical limitation, as there exists algorithms for minimizing DFAs in polynomial (in the size of the DFA and its alphabet) time [82, 134].*

2.1.3 Transducers

Formal languages provide a precise formalism for describing successful runs of a system. However, rather than only distinguishing between good runs and bad runs of a system, one may want to express the characteristics of a system from an interactive point of view, e.g., by describing how certain *inputs* to a system result in certain *outputs* from the system. A possible way to achieve this goal is by extending the previously introduced concepts to the notion of *bi-languages*. Typically, bi-languages connect words of two different domains (alphabets), which allow one to associate *input* words with *output* words. This concept leads to the general notion of *transductions*.

Definition 10 (Transduction)
Let I denote an input alphabet and O denote an output alphabet. A transduction *from I^* to O^* is a relation $T \subseteq I^* \times O^*$.*

Note that while relations are generally non-directional, we interpret transductions as a mapping *from inputs to outputs* throughout this thesis. As discussed in Section 1.1.2, this thesis focuses on deterministic transductions that follow an incremental lock-step pattern. This means, for each input symbol there exists a unique output symbol such that their respective prefixes are a valid transductions as well. Formally, a transduction T satisfies these properties iff the following statements hold:

$$\forall \langle w_a, w_o \rangle \in T : |w_a| = |w_o| \wedge \tag{2.1}$$
$$\forall \langle w_a, w_o \rangle \in T : \forall i \in \{1, \ldots, |w_a|\} : \langle w_a[,i], w_o[,i] \rangle \in T \wedge \tag{2.2}$$
$$\forall w \in I^* : |\{\langle w_a, w_o \rangle \in T \mid w = w_a\}| \leq 1 \tag{2.3}$$

where Equations (2.1) and (2.2) ensure that (all prefixes of) input words are mapped to output words of equal length and Equation (2.3) ensures (input-) determinism. Technically, the empty transduction $\langle \varepsilon, \varepsilon \rangle$ may be considered here as well, but it does not hold any significant information that distinguishes a system from another one.

Behaviorally, these types of transductions can be used to model *dialog* systems in which the system interaction is based on an alternating sequence of input actions (from the user

or client) and output reactions (from the system). In cases where the input language resembles a regular formal language, these transductions can be represented by Mealy machines [121].

Definition 11 (Mealy machine)
A Mealy machine *is a tuple* $M = \langle Q, q_0, I, O, \delta, \lambda \rangle$ *where*

- Q *denotes the non-empty, finite set of states,*
- $q_0 \in Q$ *denotes the initial state,*
- I *denotes the input alphabet,*
- O *denotes the output alphabet,*
- $\delta : (Q \times I) \to Q$ *denotes the state transition function, and*
- $\lambda : (Q \times I) \to Q$ *denotes the output function.*

We define the generalized transition functions $\delta : (Q \times I^*) \to Q$ *and* $\delta : I^* \to Q$ *identical to Definition 8 and the generalized output functions* $\lambda : (Q \times I^*) \to O^*$ *and* $\lambda : I^* \to O^*$ *as*

$$\begin{aligned} \lambda(q, u \cdot v) &= \lambda(q, u) \cdot \lambda(\delta(q, u), v) && \forall q \in Q, u \in I, v \in I^*, \\ \lambda(q, \varepsilon) &= \varepsilon && \forall q \in Q, \\ \lambda(w) &= \lambda(q_0, w) && \forall w \in I^*. \end{aligned}$$

Similar to DFAs, we may interpret δ *and* λ *as relations if it is more convenient.*

Interestingly, Mealy machines merge the ideas of generators and parsers from the previous two (sub-) sections, since a Mealy machine "parses" input symbols and "generates" output symbols. Similar to DFAs, we introduce some specific properties of Mealy machines.

Definition 12 (Properties of Mealy machines)
Let $M = \langle Q, q_0, I, O, \delta, \lambda \rangle, M_1, M_2$ *denote some Mealy machines over an input alphabet* I *and an output alphabet* O.

- *We define the* size $|M|$ *of a Mealy machine* M *as the number of its states, i.e.,* $|M| = |Q|$.
- *We define the* transduction $T(M)$ *of a Mealy machine* M *as* $T(M) = \{\langle w, \lambda(w) \rangle \mid w \in I^*\}$.
- *We call a Mealy machine* M_1 equivalent *to another Mealy machine* M_2 *(denoted as* $M_1 \equiv_{Mealy} M_2$*) iff* $T(M_1) = T(M_2)$.
- *Unless specified otherwise, we consider* total Mealy machines *in this thesis, i.e., Mealy machines where the transition function and output function are defined for all possible input arguments. In a* partial Mealy machine*,* δ *or* λ *may be undefined for certain input arguments. In this case, the Mealy machine is no longer able to emit valid output symbols and the transduction process stops.*
- *We call a Mealy machine* M_1 minimal *iff there exists no other, equivalent Mealy machine* M_2 *such that* $|M_2| < |M_1|$. *Similar to DFAs, minimality also implies canonicity and the minimization algorithms for DFAs can be easily adjusted towards Mealy machines.*

Regarding expressiveness, Mealy machines are able to process regular languages similar to DFAs. This is easily shown by transforming a Mealy-based transduction into a prefix-closed regular language. Two commonly used transformations involve either the

synchronous transduction over the cartesian product of the input alphabet and the output alphabet or the alternating transduction over the union of the two alphabets. For example, let $\lambda(a_1 \cdot a_2) = o_1 \cdot o_2$ be a Mealy-based transduction. The corresponding DFAs may either accept the word $\langle a_1, o_1 \rangle \cdot \langle a_2, o_2 \rangle$ (synchronized) or $a_1 \cdot o_1 \cdot a_2 \cdot o_2$ (alternating). By transforming a Mealy machine on a per-transition basis, the described transduction can be directly embedded in a (prefix-closed) regular language.

However, note that the reverse process, i.e., representing a synchronous or alternating transduction of a prefix-closed DFA via a Mealy machine, may introduce a kind of semantic gap. A prefix-closed DFA may at some point reject an input symbol which would terminate the transduction. In contrast, (total) Mealy machines by definition continue to transduce input symbols until there are no more input symbols left to process. One possibility to express rejection via Mealy machines is to introduce a specific "error" output symbol and an error-sink state which is accessed when the DFA would reject the word.

2.1.4 Generalizations

While DFAs and Mealy machines are tailored towards different use-cases, they share a lot of concepts that are generalized to labeled transition systems (LTSs) with outputs in the following. Specifically for techniques from the field of model verification and model-based testing (MBT), this allows for a unified presentation of concepts that can be applied to both DFAs and Mealy machines, respectively.

Definition 13 (LTS (with outputs))
An LTS with outputs is a tuple $\mathbb{L} = \langle Q, q_0, I, D, \delta, \lambda \rangle$ *where*

- Q *denotes the non-empty set of states,*
- $q_0 \in Q$ *denotes the initial state,*
- I *denotes the input alphabet,*
- D *denotes the output domain,*
- $\delta : (Q \times I^*) \to Q$ *denotes the state transition function, and*
- $\lambda : (Q \times I^*) \to D$ *denotes the output function.*

Similar to DFAs and Mealy machines, we generalize the transition function and the output function to words over I *via*

$$\delta(w) = \delta(q_0, w) \qquad \forall w \in I^*,$$
$$\lambda(w) = \lambda(q_0, w) \qquad \forall w \in I^*,$$

and may use the relation-based interpretation of these functions if convenient.

Note that the above definition of LTSs (with outputs) only considers (input-) deterministic transition systems. While many verification algorithms and testing algorithms are able to handle non-deterministic systems as well, this thesis only uses LTSs (with outputs) to generalize the (hierarchical) behavior of DFAs and Mealy machines which are deterministic by definition. However, Definition 13 does allow for infinite state LTSs which is particularly useful when talking about (deterministic) context-free systems.

Specifically for the DFA-based and Mealy machine-based procedures discussed in this

thesis, the following two definitions give a direct construction for the respective LTSs (with outputs).

Definition 14 (DFA-based LTS)
Let $A = \langle Q^A, q_0^A, I^A, Q_F^A, \delta^A \rangle$ *denote a DFA. The* LTS-based interpretation of A *is given by* $\mathbb{L}_{DFA} = \langle Q, q_0, I, D, \delta, \lambda \rangle$ *such that*

- $Q = Q^A$,
- $q_0 = q_0^A$,
- $I = I^A$,
- $D = \{true, false\}$,
- $\delta = \delta^A$, *and*
- $\lambda(q, w) = \begin{cases} true & \text{if } \delta^A(q, w) \in Q_F^A \\ false & \text{otherwise} \end{cases}$

  for all $q \in Q, w \in I^*$.

Definition 15 (Mealy-based LTS)
Let $M = \langle Q^M, q_0^M, I^M, O^M, \delta^M, \lambda^M \rangle$ *denote a Mealy machine. The* LTS-based interpretation of M *is given by* $\mathbb{L}_{Mealy} = \langle Q, q_0, I, D, \delta, \lambda \rangle$ *such that*

- $Q = Q^M$,
- $q_0 = q_0^M$,
- $I = I^M$,
- $D = (O^M)^*$,
- $\delta = \delta^M$, *and*
- $\lambda = \lambda^M$.

2.2 Model Verification

Formal languages establish a rather input-focused characterization of systems. A second, widely-used interpretation is based on Kripke transition systems (KTSs) [108] which originally omit labels but instead characterize each state via an observable set of *atomic propositions*. Often a mixture of both approaches, e.g., a labeled KTS, can be found, too. What unifies these different model types is the fact that they are transition systems which exhibit their behavior via *paths*. As a result, the verification of such models often involves a formal specification of paths that the models are required or prohibited to take. The process of model verification then checks whether there are any discrepancies between the specified paths of the requirements and the possible paths of the model.

The central idea for the verification of instrumented context-free systems in this thesis is to transform models of SPAs, SBAs, and SPMMs into other formalisms for which there already exist verification techniques. Especially, it is not the goal to develop yet another specification formalism. In order to give an intuition for such existing specification formalisms and their semantics, the following sketches the concepts of the computational tree logic (CTL) [46]. We continue with formalizing the notion of paths of LTSs.

Definition 16 (Paths)
Let I be an input alphabet and $\mathbb{L} = \langle Q, q_0, I, D, \delta, \lambda \rangle$ denote an LTS over I. A path *$\tau \in (Q \times I \times Q)^*$ is a sequence of transitions, i.e., $\tau = \langle q_1, a_1, q_2 \rangle \cdot \langle q_2, a_2, q_3 \rangle \cdot \ldots$, such that $\langle q_j, a_j, q_{j+1} \rangle \in \delta$ for all j. We define the set of paths of maximum length originating in $q \in Q$ as Paths(q). Depending on $\mathbb{L}$, paths of maximum length may either be finite or infinite.*

Note that CTL formulae are originally meant to specify paths of KTSs. However, in the context of formal languages, *labeled* paths are the more popular way to express behavior. As a result, we look at a slightly adjusted version of CTL for LTSs, that includes *actions* and omits atomic propositions. Its concepts are similar to action-based CTL (ACTL) by Nicola et al. [132], however, its syntax is more akin to the Hennessy-Milner logic (HML) [76] or the modal μ-calculus [107]. This adjustment is chosen as the syntax returns in Chapter 4, where the language verification of SPAs, SBAs, and SPMMs is presented on the basis of a model checker for (alternation-free) modal μ-calculus formulae.

Definition 17 (Syntax of CTL formulae with actions)
Let I be an input alphabet and $a \in I$. CTL formulae with actions over I consist of state formulae *and* path formulae*, where state formulae are constructed according to the grammar*

$$\Phi \longrightarrow \mathit{true} \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \mathbf{A}\,\varphi \mid [a]\Phi$$

and path formulae according to the grammar

$$\varphi \longrightarrow \neg\varphi \mid \mathbf{X}\,\Phi \mid \Phi_1\,\mathbf{U}\,\Phi_2.$$

From the above syntax, we can derive the dual operators as well as the "finally" and "globally" path operators.

$$\begin{aligned}
\mathbf{E}\,\varphi &= \neg\,\mathbf{A}\neg\varphi && \textit{(exists)},\\
\langle a\rangle\Phi &= \neg[a]\neg\Phi && \textit{(diamond a)},\\
\mathbf{EF}\,\Phi &= \mathbf{E}(\mathit{true}\,\mathbf{U}\,\Phi) && \textit{(exists finally)},\\
\mathbf{AF}\,\Phi &= \mathbf{A}(\mathit{true}\,\mathbf{U}\,\Phi) && \textit{(always finally)},\\
\mathbf{EG}\,\Phi &= \neg\,\mathbf{AF}\,\neg\Phi && \textit{(exists globally), and}\\
\mathbf{AG}\,\Phi &= \neg\,\mathbf{EF}\,\neg\Phi && \textit{(always globally)}.
\end{aligned}$$

The basic boolean constants and boolean operators follow the commonly known semantics of propositional logic including the standard derivations of *false*, $\vee$, $\Rightarrow$, and $\Leftrightarrow$. The temporal modalities **A** (pronounced "all") and **E** (pronounced "exists") require all paths (respectively, that there exists at least one path) originating in a given state to satisfy the path formula φ. The input modalities $[a]$ (pronounced "box a") and $\langle a\rangle$ (pronounced "diamond a") serve as "filters" compared to the generic quantifiers and require all a-successors (respectively, that there exists at least one a-successor) of a given state to satisfy the formula Φ. The semantics are formalized as follows:

Definition 18 (Satisfaction relation of CTL formulae with actions)
Let I be an input alphabet and $\mathbb{L} = \langle Q, q_0, I, D, \delta, \lambda \rangle$ denote an LTS over I. Let Φ, Φ_1, Φ_2 be CTL state formulae and φ be a CTL path formula. For a state $q \in Q$, we have the satisfaction relation

$$
\begin{aligned}
q &\models \mathit{true} && \Leftrightarrow \mathit{true},\\
q &\models \neg\Phi && \Leftrightarrow q \not\models \Phi,\\
q &\models \Phi_1 \wedge \Phi_2 && \Leftrightarrow q \models \Phi_1 \wedge q \models \Phi_2,\\
q &\models \mathbf{A}\,\varphi && \Leftrightarrow \forall \tau \in \mathit{Paths}(q)\colon \tau \models \varphi, \textit{ and}\\
q &\models [a]\Phi && \Leftrightarrow \forall \langle q, a, q_1 \rangle \cdot \ldots \in \mathit{Paths}(q)\colon q_1 \models \Phi.
\end{aligned}
$$

For a non-empty path $\tau = \langle q_1, a_1, q_2 \rangle \cdot \langle q_2, a_2, q_3 \rangle \cdot \ldots$ we have the satisfaction relation

$$
\begin{aligned}
\tau &\models \neg\varphi && \Leftrightarrow \tau \not\models \varphi,\\
\tau &\models \mathbf{X}\,\Phi && \Leftrightarrow q_2 \models \Phi, \textit{ and}\\
\tau &\models \Phi_1\,\mathbf{U}\,\Phi_2 && \Leftrightarrow \exists j \geq 1\colon (q_j \models \Phi_2 \wedge (\forall 1 \leq k < j\colon q_k \models \Phi_1)).
\end{aligned}
$$

We say that an LTS $\mathbb{L}$ satisfies a CTL state formula with actions Φ iff $q_0 \models \Phi$.

For details about the actual model checking algorithm, the reader may refer to the respective literature as the implementation of a model checker itself is beyond the scope of this thesis. CTL formulae with actions may be directly transformed into ACTL formulae of [132]. The diamond operator $\langle a \rangle \Phi$ corresponds to the existentially quantified action formula $\mathbf{EX}_a\,\Phi$, whereas as the box operator $[a]\Phi$ corresponds to the universally quantified action formula $\mathbf{AX}_a\,\Phi$. The verification thereof is discussed in [132]. Alternatively, one can transform the CTL formulae with actions into the (alternation-free) modal μ-calculus, where the box operator and the diamond operator exist with identical semantics and the existential quantifier and the universal quantifier correspond to the smallest and greatest fix-point operators, respectively. See, e.g., [35] for further transformations.

2.3 Model-Based Testing

The field of MBT covers a plethora of techniques with different goals and purposes [36, 111]. In this thesis, we are specifically interested in the sub-field of *conformance testing* [68, 111]. Conformance testing takes a known (white-box) model and tries to verify whether an unknown (black-box) implementation behaves equivalent to the model. Typically, this involves a set of tests that are executed on the implementation whose responses are then compared to the expected outputs of the model. Sometimes *monitoring*, i.e., the analysis of logged traces in production, may be used for this purpose as well, although conceptually it falls more into the field of *runtime verification*.

In this thesis, we focus our notion of conformance testing on DFA-based systems and Mealy-based systems. Therefore, we formalize the concept of conformance testing as follows:

Definition 19 (Conformance test)
Let I denote an input alphabet, D denote an output domain, and $\psi \in \{DFA, Mealy\}$ denote a model type. Let $\mathbb{L}^m_\psi = \langle Q^m, q^m_0, I, D, \delta^m, \lambda^m \rangle$ denote a ψ-based LTS model over I and $\mathbb{L}^i_\psi = \langle Q^i, q^i_0, I, D, \delta^i, \lambda^i \rangle$ denote a ψ-based LTS implementation over I. A conformance test *for $\mathbb{L}^m_\psi$ is a set of test words $CT(\mathbb{L}^m_\psi) \subseteq I^*$ such that*

$$(\forall w \in CT(\mathbb{L}^m_\psi) \colon \lambda^m(w) = \lambda^i(w)) \Rightarrow \mathbb{L}^m_\psi \equiv_\psi \mathbb{L}^i_\psi.$$

Note that, in general, it is not possible to construct a finite conformance test. Even for simplest formalisms such as DFAs, the true black-box equivalence problem is impossible to solve [126]. It is easy to see, how for any finite conformance test, one can construct an (unknown) implementation that behaves equivalent to the model for the test words, but in-equivalent for any other word.

As a consequence, the correctness and completeness of conformance testing algorithms often need to be analyzed under certain assumptions about the unknown implementation. For example, if one assumes an upper bound on the number of states of $\mathbb{L}^i_\psi$, one can simply traverse all possible transitions of $\mathbb{L}^i_\psi$ and verify the reached states afterwards. These assumptions usually dictate the structure of a conformance test as well. For example, if the implementation supports a kind of reset mechanism, the conformance test may consist of multiple independent test cases whose evaluation is separated by intermediate resets. If the implementation is strongly connected, i.e., every state is connected to every other state (possibly via a sequence of multiple transitions), the conformance test may consist of only a single test word and does not require the implementation to support a reset mechanism.

The discussions of Chapter 5 try to treat conformance testing as general as possible. The core idea of conformance testing SPAs, SBAs, and SPMMs is to conclude global conformance from the conformance of the respective procedural components. As a result, whichever assumptions about the procedures of an implementation hold in a certain scenario, dictate the structure, the correctness, and the completeness of the global conformance test.

To give an intuition for the computation of conformance tests, the remainder of this section briefly sketches the ideas of the W-method [44] whose ideas return in Chapter 5. The W-method computes a set of multiple test words which require the black-box implementation to support a reset mechanism. In case of software systems and hardware systems this is often relatively easy to achieve by means of restarting or rebooting the system or by using separate logical instances, e.g., a different account per test. Furthermore, the W-method requires an upper bound on the number of states of the implementation in order to reason about the correctness and completeness of the constructed conformance test. In the following, we assume the even stronger property that the model and implementation have the same amount of states, which allows for a simplified presentation of the W-method.

The core idea of the W-method revolves around two sets: a *transition cover set* and a *characterizing set*. A transition cover set contains input words that traverse each transition of a transition system. It can be easily constructed from a *state cover set*, i.e., a set

containing input words that traverse each state of a transition system, and concatenating each element with an input symbol. A characterizing set is a set CS such that for two arbitrary states s_1, s_2 of a transition system, CS contains at least one input word w for which s_1 and s_2 exhibit different observable behavior when processing w. By computing the two sets based on the model and constructing the cartesian product of the two sets, the W-method ensures that each transition of the unknown implementation transfers the system into a state that is behaviorally equivalent to its counterpart in the model, ensuring conformance. Note that the W-method considers ε an element of the transition cover set as well, in order to validate the initial state of an automaton.

The work on SBAs and SPMMs requires a more generalized notion of state cover sets and transition cover sets that allows for a restriction of the set of eligible input symbols and that works with potentially partial systems. Therefore, we first look at the notion of *reachable states* and then discuss the respective sets.

Definition 20 (Reachable states)
Let I be an input alphabet and $\mathbb{L} = \langle Q, q_0, I, D, \delta, \lambda \rangle$ be an LTS over I. The set of reachable states *of $\mathbb{L}$ over $I' \subseteq I$ is a maximal set $Reach_{I'}(\mathbb{L}) \subseteq Q$ such that*

$$\forall q \in Reach_{I'}(\mathbb{L}) \colon \exists w \in I'^{*} \colon \delta(q_0, w) = q,$$

i.e., $Reach_{I'}(\mathbb{L})$ contains all states of $\mathbb{L}$ that are reachable from q_0 via symbols of I'.

Definition 21 (State cover set)
Let I be an input alphabet and $\mathbb{L} = \langle Q, q_0, I, D, \delta, \lambda \rangle$ be an LTS over I. A state cover set *for $\mathbb{L}$ over $I' \subseteq I$ is a (minimal) set $SCS_{I'}(\mathbb{L}) \subseteq I'^{*}$ such that*

$$\forall q \in Reach_{I'}(\mathbb{L}) \colon \exists w \in SCS_{I'}(\mathbb{L}) \colon \delta(q_0, w) = q,$$

i.e., for every reachable state there exists a word in $SCS_{I'}(\mathbb{L})$ that reaches the state.

Definition 22 (Transition cover set)
Let I be an input alphabet and $\mathbb{L} = \langle Q, q_0, I, D, \delta, \lambda \rangle$ be an LTS over I. A transition cover set *for $\mathbb{L}$ over $I' \subseteq I$ is a set $TCS_{I'}(\mathbb{L}) \subseteq I'^{*}$ such that*

$$TCS_{I'}(\mathbb{L}) = SCS_{I'}(\mathbb{L}) \cdot I',$$

i.e., for every outgoing transition (with respect to I') of a reachable state there exists a word in $TCS_{I'}(\mathbb{L})$ whose last symbol traverses the transition.

Note that the above sets can be easily computed via, e.g., a breadth-first or depth-first reachability analysis that is restricted to the allowed symbols. While state cover sets do not need to be minimal, any redundant access sequence does not hold any additional value. Therefore, a minimal state cover set may be preferred for reasons of efficiency. If the input alphabet is not restricted, i.e., $I' = I$, the subscript may be omitted.

The definition of the characterizing set does not require any restrictions of the alphabet.

Definition 23 (Characterizing set)
Let I be an input alphabet and $\mathbb{L} = \langle Q, q_0, I, D, \delta, \lambda \rangle$ be an LTS over I. A characterizing set *for $\mathbb{L}$ is a (minimal) set $CS(\mathbb{L}) \subseteq I^*$ such that*

$$\forall q_1, q_2 \in Q : \exists w \in CS(\mathbb{L}) : \lambda(q_1, w) \neq \lambda(q_2, w),$$

i.e., for every pair of states there exists a word in $CS(\mathbb{L})$ that exposes an observable inequivalence.

In a minimal LTS, there exist no equivalent states, i.e., states that produce the same output for all possible input sequences. As a result, for every pair of states, there must exist at least one input sequence that separates the two states via their observable behavior. Therefore, the union of all pair-wise separating sequences constitutes a straightforward characterizing set. Again, the characterizing set does not need to be minimal, but redundant elements do not increase its characterizing property any further.

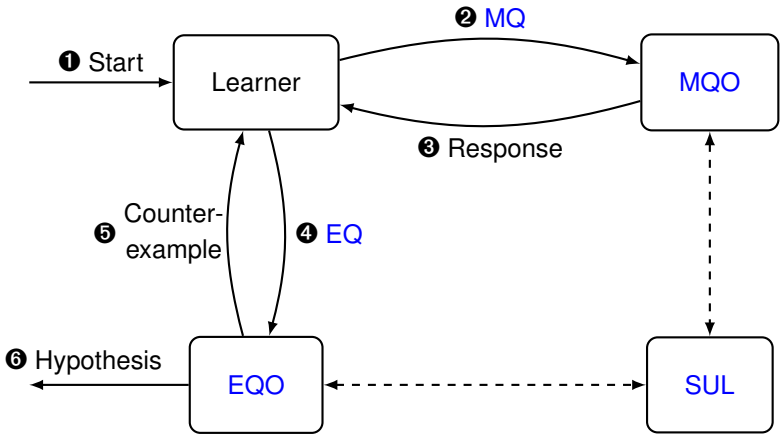
2.4 Active Automata Learning

Active automata learning (AAL) describes the process of inferring an automaton-based model on the basis of observations from actively querying a system. In [15], Angluin introduces the fundamental concepts of this process which many of today's AAL algorithms still build upon: the minimally adequate teacher (MAT) framework. This framework provides the learning algorithm with a *teacher* that grants access to two kinds of queries: membership queries (MQs) and equivalence queries (EQs).

Membership Queries and Membership Query Oracles An MQ is a word over an (input) alphabet for which the learner wants to query the system's behavior. Access to the system under learning (SUL) is formalized via a membership query oracle (MQO) which receives an MQ and responds with the system's behavior. Conceptually, MQOs can be seen as a function $mqo : I^* \to D$ for some input domain I and some output domain D. This thesis focuses on the boolean output domain for acceptor-based learning processes, e.g., for DFAs, and word-based output domains over some output alphabet for transduction-based learning processes, e.g., for Mealy machines. However, more complex output domains are possible (cf. Section 8.3). Similar to MBT, MQOs can be implemented via *testing*, i.e., translating the abstract input symbols to actual system inputs and executing the respective queries on the SUL while returning the observed response from the system.

Remark 1
In her seminal work, Angluin [15] uses AAL to infer formal languages. Hence, the term membership query *is chosen because it answers the question "is the word a* member *of the unknown language?". Despite the fact that many modern algorithms deal with more complex output domains, the term "membership query" prevails throughout literature. Similarly, the term is used interchangeably throughout this thesis for acceptor-based inference and transduction-based inference as well.*

Figure 2.1
AAL in the MAT framework.



Equivalence Queries and Equivalence Query Oracles An EQ consists of a hypothesis model for which the learner wants to know whether it is equivalent to the unknown target system. This test is performed by an equivalence query oracle (EQO). As discussed in Definitions 9 and 12, the notion of equivalence may differ depending on the targeted automaton type of the inference process and the methods for determining (in-) equivalence may vary from case to case. If the EQO detects an in-equivalence, a *counterexample* $ce \in I^*$ is returned which exposes mismatching behavior between the hypothesis model and the SUL.

2.4.1 Learning Loop

Many AAL algorithms based on the MAT framework often use MQs and EQs in a loop-like fashion, alternating between an *exploration phase* and a *verification phase*. This structure is sketched in Figure 2.1.

Exploration Phase

The learning loop starts with the exploration phase (❶) in which the learner posts MQs (❷) to obtain information about the unknown system. Via the MQO, these queries are delegated to the SUL and its responses, e.g., successful termination, emitted outputs, etc., are returned to the learning algorithm (❸) which constructs an (initial) hypothesis consistent with the observations.

Verification Phase

Once the learner has constructed a (tentative) hypothesis from the observations, it poses an EQ to the EQO (❹). The EQO may then interact with the SUL to potentially detect in-equivalent behavior. If the EQO determines that the hypothesis and the SUL are not equivalent, it returns a counterexample to the learner (❺), which the leaner may use to

refine the hypothesis model, e.g., by adding new states to the hypothesis model. The newly discovered characteristics often require the learner to explore new properties about the system which leads to another iteration of the exploration phase with new MQs. Otherwise, if the EQO determines that the hypothesis and target system are equivalent, the learning process terminates and the last constructed hypothesis is returned (❻).

Recall from Section 2.3 that the general black-box equivalence problem is impossible to solve even for simplest model types such as DFAs [126]. This makes AAL in general neither a complete nor a correct process because the EQO may not detect all in-equivalencies. However, there exist plenty of success stories of applying AAL in practice (cf. Section 8.3).

2.4.2 Characteristics of Learning Algorithms

The two major challenges of AAL algorithms (and what is mostly used as distinguishing features between them) are

1. to efficiently store the information about the observed behavior which directly affects what and how many queries are posed during the exploration phase and
2. to efficiently analyze counterexamples from EQs in order to extract only relevant information for refining the current hypothesis.

As a result, most MAT-based AAL algorithms are presented, analyzed, and compared with each other by how they tackle these two challenges. Therefore, Chapter 6 focuses on how the learners for SPAs, SBAs, and SPMMs access the information from the SUL during the exploration phase and how the information from counterexamples during verification are processed to refine the current hypotheses.

CHAPTER 3

Instrumented Context-Free Systems

This chapter presents the model types of systems of procedural automata (SPAs), systems of behavioral automata (SBAs), and systems of procedural Mealy machines (SPMMs). It establishes the formal definitions and semantics of the formalisms and elaborates on several properties that are used by the techniques of the remaining chapters. Specifically for SPAs and SBAs, this includes a monitor-based interpretation which allows for additional use-cases in the context of model-based testing (MBT).

3.1 Motivation

Consider the exemplary context-free grammar (CFG) G_{palin} in Example 2.

Example 2 (A CFG for palindromes over the letters a, b, c [61])
Let $G_{palin} = \langle N, T, P, S \rangle$ *denote a CFG with*

- $N = \{F, G\}$,
- $T = \{a, b, c\}$,
- P *is given by the following set of production rules in Backus-Naur form (BNF)*
  `F -> a | a F a | b | b F b | G | ε`
  `G -> c | c G c | F`
  and
- $S = F$.

It is easy to see that the context-free language (CFL) induced by G_{palin} consists of palindromes[2] over the letters a, b, c. However, one should note a special property of G_{palin}: It uses two distinct non-terminal symbols to "split the work" of emitting terminal symbols. The production rules of the non-terminal symbol F emit the terminal symbols a and b whereas the production rules of the non-terminal symbol G emit the terminal symbol c. For the induced language, this property is insignificant because one can easily construct a second CFG for palindromes over a, b, c that only uses a single non-terminal symbol. However, in the context of model-based quality assurance (MBQA), where models should reflect properties of a system, this deliberate decision of decomposing the system into two modular components is an essential piece of information about the structure and the semantics of a system.

[2] Palindromes are words that are identical when read left-to-right and right-to-left.

Its importance is linked to the way we continue to interpret context-free systems in this thesis: With non-terminals we associate the notion of *procedural invocations*, i.e., delegating the flow of execution to some independent (possibly recursive) component upon whose return the local work continues, whereas the *local work* is represented by terminal symbols.

Consider the production rule "`F -> a F a`" from Example 2. It states that a valid *run* of *procedure F* may consist of three subsequent actions: First, the procedure executes the terminal action a. After that, it calls another procedure (here a recursive call to itself) and delegates its execution to the invoked procedure. After returning from the call, it performs a final terminal action a before terminating itself. Being able to assign such behaviors to individual components and procedures of a system is paramount for a precise and sound application of MBQA.

Remark 2
Due to the established connection between non-terminals of a CFG and the concept of invoking procedures, we use the terms "context-free system" and "procedural system" interchangeably for the remainder of this thesis.

3.1.1 Instrumentation

A major challenge for accessing these internal characteristics of CFLs is that non-terminal symbols are not observable in the induced language. Techniques such as MBT or active automata learning (AAL) which rely on the *observable* behavior of a (black-box) system cannot distinguish between a system like in Example 2 and a system using only a single non-terminal or procedure.

To tackle this issue, this thesis proposes an *instrumentation* that makes the start and the end of procedural invocations observable. Intuitively, this is realized by modifying each procedure to perform an observable *call* action at the start of a procedural invocation and to perform a second observable *return* action before its termination. Taking Example 2 as a reference, Example 3 shows such an instrumentation.

Example 3 (An instrumented CFG of palindromes over the letters a, b, c)
Let $G_{palin-inst} = \langle N, T, P, S \rangle$ *denote a CFG with*

- $N = \{\bar{F}, \bar{G}\}$,
- $T = \{a, b, c, F, G, R\}$,
- *P is given by the following set of production rules in BNF*
  `F̄-> F a R | F a F̄ a R | F b R | F b F̄ b R | F Ḡ R | F R`
  `Ḡ-> G c R | G c Ḡ c R | G F̄ R`
  and
- $S = \bar{F}$.

Each non-terminal symbol X of the original grammar is now interpreted as a new terminal symbol and a distinct terminal return symbol R is introduced. Furthermore, the production rules have been updated according to a pattern known from *bracketed languages* [70]: Each production rule is prepended by the *call* symbol of the respective non-

terminal and the new return symbol is appended to each production rule indiscriminately. A generalization of this instrumentation is formalized in Definition 24.

Definition 24 (Instrumentation)
Let $G_1 = \langle N_1, T_1, P_1, S_1 \rangle$ denote an arbitrary CFG and $r \notin T_1$ denote a distinct return symbol. We define the instrumentation $G_2 = \langle N_2, T_2, P_2, S_2 \rangle$ *of G_1 as follows:*

- $N_2 = \{\bar{X} \mid X \in N_1\}$,
- $T_2 = N_1 \uplus T_1 \uplus \{r\}$,
- $P_2 = \{\langle \overline{LHS}, LHS \cdot RHS_{N_1 \mapsto \bar{N}_1} \cdot r \rangle \mid \langle LHS, RHS \rangle \in P_1\}$, *and*
- $S_2 = \bar{S}_1$.

where $w_{Z \mapsto \bar{Z}}$ represents the word w in which every symbol $z \in Z$ has been mapped to $\bar{z}$. From the construction of P_2, it directly follows that G_2 is context-free, too.

Instrumentation is a crucial mechanism for exposing key structural properties of a system. One can easily see by looking at the induced language of the instrumented system of Example 3 that the separation of work between the two involved procedures now becomes observable. Previous terminal symbols a and b are "nested" in matching pairs of observable symbols F and R whereas as the symbol c is "nested" in pairs of observable symbols G and R.

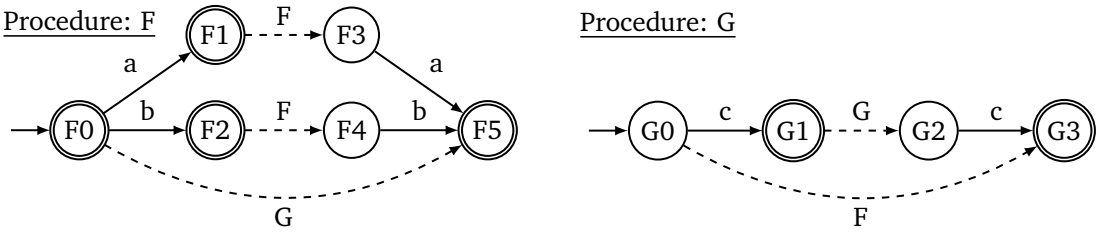
At first sight, such an instrumentation adds an additional challenge when using MBQA in practice. Besides the regular translation of abstract alphabet symbols to concrete systems actions, the instrumentation requires the system to support additional and potentially not originally available interactions. However, instrumentation is a well-established concept in software and hardware engineering so that there exists a variety of tools and frameworks to implement these modifications. Furthermore, there exist certain application domains where such information is an inherent part of system semantics which may be exploited for free. Chapter 9 discusses the technical aspects of providing the proposed instrumentation in practice as well as some fruitful application domains. In the following, we assume to have a functional instrumentation available for the ease of presentation.

3.2 SPAs

SPAs aim at providing an efficient and intuitive model for describing context-free systems. The core idea of SPAs is to represent the production rules of a CFG via a set of deterministic finite acceptors (DFAs). For each non-terminal symbol of the grammar, there exists a corresponding DFA that accepts the language of right-hand sides of the respective production rules. Therefore, each DFA represents an individual procedure that may contain (potentially recursive) calls to other procedures. To give an intuition for this representation, consider Figure 3.1 that shows such a DFA-based representation of the system of Example 2.

This representation has several major advantages: First, it is easy to understand. Automaton-based structures often closely align with the internal structure of software systems and hardware systems and allow one to visually outline important areas of a system. Compared to the rather flat structure of production rules, this boosts the

Figure 3.1 (from [61])
Two DFAs accepting the right-hand sides of the respective production rules of G_{palin} of Example 2. Sink states and corresponding transitions are omitted for readability.



understanding of system semantics. Furthermore, the semantics of SPAs (cf. Section 3.2.1) follow the standard copy-rule semantics known from the expansion of CFGs. This allows procedures to be seen in isolation, irrespective of the context in which they are executed. In comparison, related formalisms that also deal with context-free structures (cf. Chapter 7), use joint, stack-based execution semantics, which implicitly add a global state that one has to continuously keep track of.

Second, this representation is complete, i.e., it is able to cover the set of all CFLs. From Definitions 4 and 6 it follows that for each CFG there can only exist a finite amount of finite production rules and therefore only a finite amount of finite right-hand sides for each non-terminal. Since every finite language of finite words is regular, the production rules of a non-terminal are representable by a DFA. Due to the equivalence of regular languages and regular expressions [83], a DFA-based representation even supports syntactic sugar like the extended Backus-Naur form (EBNF) [92, 173].

Lastly, this representation supports a canonical form. As DFAs (and in case of SPMMs, Mealy machines) support canonical representations themselves, the aggregation of these automata yields a canonical representation, too. This proves useful for various techniques from the field of verification, testing, and learning. Using the piecewise application of minimization algorithms [82, 134] for the individual automata, directly allows one to construct minimized aggregations which remove potentially hard to detect redundancies of the CFG-based representation.

As discussed in Section 3.1.1, exposing the internal structure of a context-free system via observable behavior is not possible because the language of a system only includes terminal symbols. This problem transfers to the DFA-based interpretation as well. To decide whether a procedural execution such as "`a F a`" is successful, the respective DFA needs to parse the non-terminal symbol F. To address this issue, the language definition of SPAs directly includes the proposed instrumentation which allows for a transfer of these information between global interpretations and local interpretations of words.

Prior to detailing the semantics of SPAs, we continue with the introduction of some additional notation first. In order to better distinguish between the roles of each alphabet symbol, the notion of an *SPA input alphabet* partitions the set of input symbols according to their effect on the system.

Definition 25 (SPA input alphabet [61])
An SPA input alphabet *is a disjoint union* $\Sigma = \Sigma_{call} \uplus \Sigma_{int} \uplus \{r\}$ *where*

- Σ_{call} *denotes the* call alphabet*,*
- Σ_{int} *denotes the* internal alphabet*, and*
- r *denotes the* return symbol*.*

We write $\Sigma_{proc} = \Sigma_{call} \uplus \Sigma_{int}$ *to denote the* procedural (sub-) alphabet *of* Σ*.*

Intuitively, call symbols represent symbols that correspond to calls to other procedures. In the context of plain CFGs (cf. Example 2), these symbols correspond to the non-terminal symbols of the grammar. Internal symbols represent the atomic actions of a procedure and correspond to the terminal symbols of a CFG. The return symbol corresponds to the artificial symbol that is appended to each production rule during instrumentation. For the exemplary systems in Examples 2 and 3 the corresponding SPA input alphabet is given by $\Sigma = \{F, G\} \uplus \{a, b, c\} \uplus \{R\}$.

In the following, a special markup token is used in order to better distinguish between a local, procedural interpretation of a word and a global, instrumented interpretation of a word. We use $\widehat{\ }$ to denote the procedural interpretation of a symbol, word, or alphabet and add (or remove) this markup token to dynamically switch to the procedural (or global) interpretation of a symbol, word or alphabet. Note that this token is only used for reasons of clarity and does not change or transform the actual input symbols. We continue with the formalization of *procedural automata* and their aggregation towards *SPAs*.

Definition 26 (Procedural automaton [61])
Let Σ *be an SPA input alphabet and* $c \in \Sigma_{call}$ *denote a procedure. A* procedural automaton *for procedure* c *is a DFA* $P^c = \langle Q^c, q_0^c, \widehat{\Sigma}_{proc}, Q_F^c, \delta^c \rangle$*.*

Definition 27 (SPA [61])
Let Σ *be an SPA input alphabet with* $\Sigma_{call} = \{c_1, \ldots, c_n\}$*. An* SPA *over* Σ *is a tuple* $S = \langle P^{c_1}, \ldots, P^{c_n} \rangle$ *such that for each call symbol there exists a corresponding procedural automaton. The* initial procedure *of* S *is denoted as* $c_0 \in \Sigma_{call}$*.*

3.2.1 Semantics

For formally specifying the semantics of SPAs, structural operational semantics (SOS) [139] is used. The SOS formalism allows one to specify operational semantics via a set of rules that transform some form of application state. These state transformations can optionally be guarded by generic *constraints* and so-called *control components* that may also be transformed by the SOS rules. This formalism is chosen because on the one hand, it is a very intuitive yet powerful way of specifying behavioral properties which makes formulating and understanding proofs easier. On the other hand, it is implementation-independent which means that in practice, SPAs may be implemented with different techniques such as push-down automata, graph-transformations, or grammar-based interpretations as long as they are behaviorally equivalent to the SOS specification.

Typically, SOS-based proofs follow the structure of a deduction tree. An initial configuration represents the root of a tree and each node branches into several child-nodes

depending on which transformation rules are applicable. The leaves of this tree then represent the inferable statements. However, this thesis uses a more familiar interpretation based on transition systems. Here, the configurations consisting of states and control components form the nodes of a transition system which are connected with each other via (guarded) transitions that correspond to the (guarded) transformation rules of the SOS formalism. Whether one SOS configuration can be deducted from another SOS configuration then transforms into a reachability problem. Hence, a deduction tree of SOS rules is often referred to as an *SOS system* in the following.

Notation-wise, the classic SOS formalism is extended to support a notion of *emitting* symbols in order to specify a generative mechanism (cf. Section 2.1.1). The statement

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

for some states s_1, s_2 and some control components σ_1, σ_2 denotes that this transformation (if applicable) *emits* a symbol o. This notation is generalized 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 emitted symbols yields w.

For defining the semantics of SPAs via SOS, we first look at the notion of an *SPA stack* which is used to model the control components of the SOS rules and then continue with the formal definition of the (instrumented) language of an SPA.

Definition 28 (SPA stack domain/configuration [61])
Let Σ be an SPA input alphabet. We define $\Gamma_{SPA} = \widehat{\Sigma}^* \uplus \{\bot\}$ as the SPA stack domain *with $\bot$ as the unique* bottom-of-stack symbol. *We use $\bullet$ to denote the stacking of elements of Γ_{SPA} where writing elements from left to right displays the stack from top to bottom and we write $ST(\Gamma_{SPA})$ to denote the set of all possible* stack configurations.

Definition 29 (Language-SOS of an SPA [61])
Let Σ be an SPA input alphabet and S be an SPA over Σ. Using tuples from $\widehat{\Sigma}^* \times ST(\Gamma_{SPA})$ to denote a system configuration, we define three kinds of SOS transformation rules:

1. *call-rules:*

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

   for all $\widehat{c} \in \widehat{\Sigma}_{call}, \widehat{v} \in \widehat{\Sigma}^*, \widehat{w} \in \widehat{\Sigma}^*_{proc}, \sigma \in ST(\Gamma_{SPA})$.

2. *int-rules:*

$$\frac{-}{(\widehat{a} \cdot \widehat{v}, \sigma)_{SPA} \xrightarrow{a} (\widehat{v}, \sigma)_{SPA}}$$

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

3. *ret-rules:*

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

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

The language of an SPA is then defined as

$$L(S) = \{w \in \Sigma^* \mid (\widehat{c}_0, \bot)_{SPA} \xrightarrow{w}{}^*(\varepsilon, \bot)_{SPA}\}.$$

We call a word $w \in \Sigma^*$ admissible *in* S *iff* $\exists \widehat{v} \in \widehat{\Sigma}^*, \sigma \in ST(\Gamma_{SPA})$ *such that*

$$(\widehat{c}_0, \bot)_{SPA} \xrightarrow{w}{}^*(\widehat{v}, \sigma)_{SPA}.$$

To showcase the language-SOS, Example 4 presents an exemplary run through the SOS system of the SPA $S = \langle P^F, P^G \rangle$ based on Figure 3.1, using F as the initial procedure.

Example 4 (An exemplary run of the SPA based on Figure 3.1 with initial procedure F [61])
The SOS system starts with the configuration $(\widehat{F}, \bot)_{SPA}$. *Since* $\widehat{a} \cdot \widehat{F} \cdot \widehat{a} \in L(P^F)$, *we can apply a call-rule to perform the transition*

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

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

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

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

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

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

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

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

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

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

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

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

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

Applying a ret-rule again, we get

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

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

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

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

Note how the language-SOS implicitly instruments the languages of the individual procedures of an SPA. Call-rules emit the respective call symbol when constructing the state of the invoked procedure and append a return symbol that is emitted by the ret-rule prior to returning from the procedure again. Since the runs of words of the language of an SPA need to terminate with an empty "stack" (or rather control component), this instrumentation results in a form of matchedness or nesting that is further analyzed in the following sections. Also, Section 3.2.5 shows that these semantics coincide with the classic expansion semantics of CFGs when applied to an instrumented (cf. Definition 24) system, i.e., the language-SOS is effectively a shortcut for the process of instrumentation and expansion.

Given the language semantics of SPAs, we continue with the presentation of several SPA-related properties.

Definition 30 (Properties of SPAs)
Let Σ *be an SPA input alphabet and* $S = \langle P^c, \ldots \rangle, S_1, S_2$ *denote some SPAs over* Σ.

- *We write* $|S| = \sum_{c \in \Sigma_{call}} |P^c|$ *to denote the* size *of an SPA.*
- *We call an SPA* S minimal *(with respect to* Σ*) iff*
  - *each procedural automaton* P^c *is minimal (with respect to* $\widehat{\Sigma}_{proc}$*) and*
  - $\forall c \in \Sigma_{call} \colon \exists w \in L(S), i \in \{1, \ldots, |w|\} \colon w[i] = c$.

  Note that the latter property requires all procedural automata to be reachable (by means of the language-SOS system) and describe a non-empty language.
- *We call an SPA* S_1 equivalent *to another SPA* S_2 *(denoted as* $S_1 \equiv_{SPA} S_2$*) iff* $L(S_1) = L(S_2)$.

In the following, we only consider minimal SPAs with respect to Σ unless specified otherwise. Note that for SPAs, minimality also implies canonicity because minimal DFAs are unique (up to isomorphism) and SPAs are uniquely defined by their (canonical) procedural automata.

3.2.2 Properties of Instrumented Words

The instrumentation induced by the language-SOS implies a nesting structure of call symbols and return symbols that allows for reasoning about the scope of procedural invocations. To access and characterize these structural properties, some utility functions are necessary.

Definition 31 (Call-return balance [61])
Let Σ be an SPA input alphabet. The call-return balance is a function $\beta : \Sigma^* \to \mathbb{Z}$, 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 $u \in \Sigma, v \in \Sigma^*$.

Intuitively, the call-return balance returns the number of open (unmatched) call symbols, which may be negative if there are more open (unmatched) return symbols. One the basis of the call-return balance, it is possible to classify words regarding their matchedness.

Definition 32 ((Minimally) call-, return-, and well-matched words)
Let Σ be an SPA input alphabet. We write

- *$CM(\Sigma)$ to denote the set $\{w \in \Sigma^* \mid \forall w' \in Suff(w) \colon \beta(w') \leq 0\}$ of* call-matched words*,*
- *$RM(\Sigma)$ to denote the set $\{w \in \Sigma^* \mid \forall w' \in Pref(w) \colon \beta(w') \geq 0\}$ of* return-matched words*,*
- *$WM(\Sigma)$ to denote the set $CM(\Sigma) \cap RM(\Sigma)$ of* well-matched words*,*
- *$MCM(\Sigma)$ to denote the set $\{w \in (\Sigma^* \cdot \{r\}) \mid w \in CM(\Sigma)\}$ of* minimally call-matched words*,*
- *$MRM(\Sigma)$ to denote the set $\{w \in (\Sigma_{call} \cdot \Sigma^*) \mid w \in RM(\Sigma)\}$ of* minimally return-matched words*, and*
- *$MWM(\Sigma)$ to denote the set $MCM(\Sigma) \cap MRM(\Sigma)$ of* minimally well-matched words*.*

For well-matched words, there exists an alternative, inductive characterization that is more convenient for some proofs. We may alternatively define $WM(\Sigma) \subseteq \Sigma^*$ as the smallest set such that

- *$\Sigma_{int}^* \subseteq WM(\Sigma)$,*
- *$\forall c \in \Sigma_{call}, w \in WM(\Sigma) \colon c \cdot w \cdot r \in WM(\Sigma)$, and*
- *$\forall w_1, w_2 \in WM(\Sigma) \colon w_1 \cdot w_2 \in WM(\Sigma)$.*

In order to reason about procedural scopes, it is necessary to find the matching return symbols within instrumented words. This is formalized via the notion of a *maximum well-matched suffix function*.

Definition 33 (Maximum well-matched suffix function)
Let Σ be an SPA input alphabet and $w \in \Sigma^*$. We define the maximum well-matched suffix function *$\rho_w : \mathbb{N}_{>0} \to \mathbb{N}$ as*

$$\rho_w(x) = \max\{i \in \{x-1, \ldots, |w|\} \mid w[x, i] \in WM(\Sigma)\}.$$

Note that if there exists no well-matched suffix of $w[x,]$, e.g., if $w[x] = r$, then $\rho_w(x)$ returns $x - 1$ as $w[x, x-1] = \varepsilon \in WM(\Sigma)$.

Remark 3
In [61] the definition of ρ (there called find-return function*) returns the index of the return symbol. Therefore, certain theorems from [61] that are presented in this thesis use indices that are shifted by one.*

Furthermore, an *instances set* is used to quickly access the procedural invocations occurring in an instrumented word.

Definition 34 (Instances set [61])
Let Σ be an SPA input alphabet and $w \in \Sigma^*$. We define the instances set *$Inst_w \subseteq \Sigma_{call} \times \mathbb{N}$ as*

$$Inst_w = \{\langle c, i\rangle \mid w[i] = c \in \Sigma_{call}\}.$$

3.2.3 Expansion and Projection

As Definition 29 suggests, the procedural automata of an SPA inherently define its language. This relationship is concretized via the notion of *expansion* and *projection*: Expansion describes the process of transforming a local, procedural word into a global, instrumented word, which allows one to analyze the behavior of an individual procedure on the full system. Projection describes the reverse process of projecting from a global word the individual behaviors of all procedures occurring in the global word. For the remaining chapters of this thesis, these two concepts are of crucial importance as they allow for a formalization of two essential (de-) composition properties (cf. Section 3.2.4) that are able to lift various concepts of regular systems to the context-free environment of SPAs.

We continue with looking at the notion of the *gamma expansion* first. Essential for the expansion process is the concept of so-called access sequences, terminating sequences, and return sequences.

Definition 35 (Access sequences, terminating sequences, and return sequences [61])
Let Σ be an SPA input alphabet and S be an SPA over Σ. The context of a procedure $c \in \Sigma_{call}$ (denoted as $Cont_c \subseteq \Sigma^* \times \Sigma^*$) containing the access sequences *and* return sequences *of c and the set of* terminating sequences *of c (denoted as $TS_c \subseteq \Sigma^*$) are defined as*

$$\langle as, rs\rangle \in Cont_c \wedge ts \in TS_c \iff \exists w \in L(S)\colon \exists \langle c, i\rangle \in Inst_w\colon w = as \cdot ts \cdot rs \wedge ts = w[i+1, \rho_w(i+1)].$$

Intuitively, an access sequence of a procedure c is an admissible word that transitions the language-SOS system to the start of procedure c, i.e.,

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

for all $\widehat{w} \in L(P^c)$, $\langle as, \cdot\rangle \in Cont_c$, and some matching $\sigma \in ST(\Gamma_{SPA})$. A terminating sequence represents a successful run from the start of a procedure to its end, possibly containing some nested calls to other procedures, i.e.,

$$(\widehat{w} \cdot \widehat{r}, \sigma)_{SPA} \xrightarrow{ts}{}^*(\widehat{r}, \sigma)_{SPA}$$

for some matching $\widehat{w} \in L(P^c), ts \in TS_c, \sigma \in ST(\Gamma_{SPA})$. A return sequence of a procedure c guarantees to reach the final language-SOS configuration of an SPA after accessing the procedure via the matching access sequence and successfully terminating it, i.e.,

$$(\widehat{c}_0, \bot)_{SPA} \xrightarrow{as \cdot ts}{}^{*} (\widehat{r}, \sigma)_{SPA} \xrightarrow{rs}{}^{*} (\varepsilon, \bot)_{SPA}$$

for all $\langle as, rs \rangle \in Cont_c, ts \in TS_c$, and some matching $\sigma \in ST(\Gamma_{SPA})$. In the following, as_c, ts_c, and rs_c may be used as shorthand notations for elements $\langle as, rs \rangle \in Cont_c$, and $ts \in TS_c$, respectively.

To give an example of the three kinds of sequences, Example 5 shows a possible access sequence, terminating sequence, and return sequence for procedure P^G of the SPA of Example 4. Note that the language of an SPA often contains multiple words and therefore there may exist multiple terminating sequences and pairs of access sequences and return sequences. Section 6.2.5 formalizes and exploits this fact to boost the performance of SPA-based AAL. However, for reasons of simplicity, this chapter assumes a single representative for the access sequence, terminating sequence, and return sequence of each procedure.

Example 5 (Exemplary access sequence, terminating sequence, and return sequence of an SPA word [61])
Let S be the SPA based on Figure 3.1 and let $w = F \cdot a \cdot F \cdot G \cdot c \cdot R \cdot R \cdot a \cdot R \in L(S)$. A possible access sequence, terminating sequence, and return sequence for procedure G are:

- *access sequence: $F \cdot a \cdot F \cdot G$,*
- *terminating sequence: c, and*
- *return sequence: $R \cdot R \cdot a \cdot R$.*

The process of expanding a local, procedural word then consists of a symbol-wise processing that replaces each procedural invocation with a concatenation of the respective call symbol, the respective terminating sequence, and the return symbol. Definition 36 formalizes this process.

Definition 36 (Gamma expansion [61])
Let Σ be an SPA input alphabet. The gamma expansion $\gamma \colon \widehat{\Sigma}^*_{proc} \to WM(\Sigma)$ *is defined as*

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

for all $\widehat{u} \in \widehat{\Sigma}_{proc}, \widehat{v} \in \widehat{\Sigma}^*_{proc}$.

Note that γ requires an environment which provides at least one terminating sequence (of which γ may select an arbitrary one) for every call symbol occurring in its argument. In the context of MBT (cf. Chapter 5) terminating sequences are computed on the basis of the model and in the context of AAL (cf. Chapter 6) terminating sequences are extracted from counterexamples, so that γ is always applicable. As discussed previously, the remainder of this chapter assumes that all the corresponding terminating sequences are available.

Figure 3.2 (from [61])
The gamma expansion of a local word of a procedural automaton P^p. By additionally embedding the transformed word into a context of an access sequence and return sequence, the local word can be fully transformed to a global word of the system.

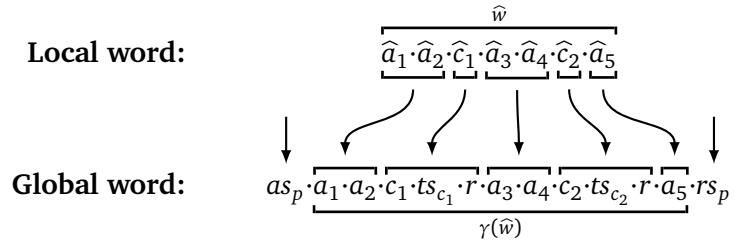


Figure 3.2 visualizes the expansion process. Note that by embedding the expanded word in a context of access sequence and return sequence, one can further establish a direct equivalence between the local behavior of the concerned procedure and the global behavior of the system (cf. Theorem 2).

The second type of transformation is denoted as *alpha projection*. Essentially, this process reverses the gamma expansion by parsing a (well-matched) instrumented word and replacing each occurrence of a nested, instrumented invocation with its procedural equivalent. The process is formalized in Definition 37.

Definition 37 (Alpha projection [61])
Let Σ be an SPA input alphabet. The alpha projection $\alpha\colon WM(\Sigma) \to \widehat{\Sigma}^*_{proc}$ *is defined as*

$$\alpha(\varepsilon) = \varepsilon,$$
$$\alpha(u \cdot v) = \begin{cases} \widehat{u} \cdot \alpha(v[\rho_v(1)+2,]) & \text{if } u \in \Sigma_{call} \\ \widehat{u} \cdot \alpha(v) & \text{if } u \in \Sigma_{int} \end{cases}$$

for all $u \in \Sigma_{proc}, v \in \Sigma^*$.

Note that α expects a well-matched word as an argument. When u is a call symbol, $\rho_v(1)+1$ represents the index of the matching return symbol. Continuing *after* this symbol guarantees that the argument to the recursive application is well-matched, too. This projection process is visualized in Figure 3.3.

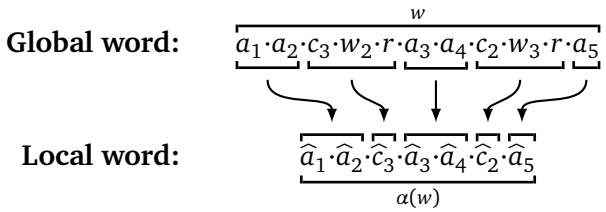
3.2.4 (De-) Composition Properties

Definitions 36 and 37 allow for the formalization of the two fundamental properties of SPAs.

Theorem 1 (Projection theorem [61])
Let Σ be an SPA input alphabet, S be an SPA over Σ, and $w \in WM(\Sigma)$ starting with c_0.

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

Figure 3.3 (from [61])
The alpha projection of an instrumented global word, which replaces nested procedural invocations so that it is interpretable in the local context of the concerned procedure.



Proof. This equivalence is based on the fact that for every emitted call symbol c of the language-SOS of S, 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 word in the language of the respective procedural automaton. For the full proof, see the *localization theorem* in [61]. □

Theorem 1 states that the semantics of an SPA are characterized by its individual procedures. Testing whether an (instrumented) word is a member of the language of an SPA can be answered equivalently by testing whether each projected run of the invoked procedures is a member of the respective procedural languages. This notion of rigorous (de-) composition allows for the implementation of various MBQA concepts for SPAs via its procedural, regular components: In the context of verification, existing techniques for regular systems may be used to verify the components of an SPA. In the context of testing (cf. Chapter 5), conformance tests for SPAs can be constructed via an aggregation of regular conformance tests for the involved procedures. In the context of AAL (cf. Chapter 6), a learner for SPAs can be constructed via an aggregation of simultaneous regular learners for the involved procedures.

However, in order to implement the proposed methods of verifying, testing, and learning SPAs on the basis of their procedures, there needs to exist the possibility to evaluate local, procedural behavior on the global, instrumented system. This process is formalized in the *expansion theorem*.

Theorem 2 (Expansion theorem [61])
Let Σ be an SPA input 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 $c \in \Sigma_{call}$, $\langle as, rs \rangle \in Cont_c$.

Proof. This equivalence is based on the fact that pairs of access sequences and matching return sequences of a procedure c provide an admissible context for arbitrary $\widehat{w} \in L(P^c)$. One can then show by induction that

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

holds for all $\widehat{w} \in L(P^c)$ and some matching $\sigma \in ST(\Gamma_{SPA})$. For the full proof, see Corollary 1 in [61]. □

Theorem 2 allows for the evaluation of local language properties of procedural automata on the global SPA system. Especially in the contexts of MBT and AAL, this property proves useful as here, only the instrumented, global system is available for executing the respective tests. Theorem 2 allows one to transform these procedural tests and evaluate them on the global SPA system.

Concluding the concepts of projection and expansion, the following corollary establishes the notion of SPA equivalence which sets the re-occurring theme for the verification, testing, and learning of SPAs.

Corollary 1 (Equivalence of SPAs)
Let Σ be an SPA input alphabet and $S_1 = \langle P_1^c, \ldots \rangle, S_2 = \langle P_2^c, \ldots \rangle$ be two SPAs over Σ. We have

$$S_1 \equiv_{SPA} S_2 \Leftrightarrow \forall c \in \Sigma_{call} \colon P_1^c \equiv_{DFA} P_2^c.$$

Proof. This is a direct consequence of Theorem 1 which characterizes the language of an SPA via the language of its procedures. Consequently, if each procedural automaton of S_1 is equivalent to its counterpart of S_2, the two SPAs describe the same language. Hence, the two systems are equivalent. □

3.2.5 Instrumentation, Expansion, and Language

This section establishes the property that for a given context-free system, the language of the respective SPA coincides with the CFL of the instrumented system. On the one hand, this underlines the soundness of the language-SOS to capture the semantics of an instrumented, context-free system. On the other hand, this establishes an alternative characterization of the semantics of SPAs that may be used in other contexts. Especially for model verification, Chapter 4 discusses an approach that is based on general context-free model checking. With establishing the above relationship, the model checker can be provided with an instrumented description of an SPA in order to verify its *language*.

In order to compare the two formalisms and reason about their equivalence, we look at an SOS-based description of CFLs. Since SPAs already serve as intuitive representations of CFGs, this description is implemented via an *expansion-SOS* for SPAs.

Definition 38 (Expansion-SOS of an SPA)
Let Σ be an SPA input alphabet and S be an SPA over Σ. We use tuples from $(\widehat{\Sigma}^* \times \{\bot\})$ to denote a system configuration, i.e., we omit any specific control components. We define two kinds of SOS transformation rules:

1. *expansion-rules:*

$$\frac{\widehat{w} \in L(P^c)}{(\widehat{c} \cdot \widehat{v}, \bot)_{Exp} \xrightarrow{\varepsilon} (\widehat{w} \cdot \widehat{v}, \bot)_{Exp}}$$

for all $\widehat{c} \in \widehat{\Sigma}_{call}, \widehat{w}, \widehat{v} \in \widehat{\Sigma}^*$.

2. *emission-rules:*

$$\frac{-}{(\widehat{a}\cdot\widehat{v},\bot)_{Exp}\xrightarrow{a}(\widehat{v},\bot)_{Exp}}$$

for all $\widehat{a}\in\widehat{\Sigma}_{int},\widehat{v}\in\widehat{\Sigma}^*$.

The set of expanded words of an SPA is then defined as

$$Exp(S)=\{w\in\Sigma^*\mid(\widehat{c}_0,\bot)_{Exp}\xrightarrow{w}{}^*(\varepsilon,\bot)_{Exp}\}.$$

It is easy to see that the expansion-SOS replicates the semantics of CFG derivations. Whenever a call symbol (representing a non-terminal symbol) is processed, the expansion-rule replaces the symbol with a production of the concerned procedure ($\widehat{w}\in L(P^c)$). All internal symbols (representing terminal symbols) are directly emitted by emission-rules. Consequently, the set of expanded words of an SPA coincides with the CFL of the corresponding CFG represented by the SPA.

For incorporating the proposed instrumentation of Definition 24, it is necessary to construct an *instrumented SPA* to reflect the modified production rules. Therefore, we first look at an *instrumented SPA input alphabet* to account for the new non-terminal symbols and terminal symbols.

Definition 39 (Instrumented SPA input alphabet)
Let Σ *be an SPA input alphabet. We write* $\bar{\Sigma}=\bar{\Sigma}_{call}\uplus(\Sigma_{call}\uplus\Sigma_{int}\uplus\{r\})\uplus\{\bot\}$ *to denote the* instrumented SPA input alphabet. *We write* $\bar{\Sigma}_{proc}=\bar{\Sigma}_{call}\uplus(\Sigma_{call}\uplus\Sigma_{int}\uplus\{r\})$ *to denote the* instrumented procedural (sub-) alphabet *of* $\bar{\Sigma}$.

For the two systems described in Examples 2 and 3, the instrumented SPA input alphabet is given by $\bar{\Sigma}=\{\bar{F},\bar{G}\}\uplus\{F,G,a,b,c,R\}\uplus\{\bot\}$. The previous call symbols F and G are now treated as internal symbols which allows the expansion-SOS to emit them as observable symbols and the new call symbols ($\bar{\Sigma}_{call}$) are used to represent the previous expansion points of the non-terminal symbols. Applying this instrumentation to a procedural automaton P^c involves three major modifications.

- Every previous c-transition needs to be replaced by a $\bar{c}$-transition to account for the new call symbols,
- a new initial state needs to be added that performs an (observable) c-transition to the old initial state, and
- all previous accepting states need to perform an r-transition into a new, single accepting state.

These steps are formalized via the notions of *instrumented procedural automata* and *instrumented SPAs*.

Definition 40 (Instrumented procedural automaton)
Let Σ *be an SPA input alphabet and* $P^c=\langle Q,q_0,\widehat{\Sigma}_{proc},Q_F,\delta\rangle$ *be a procedural automaton for some* $c\in\Sigma_{call}$. *The* instrumented procedural automaton *is a DFA* $\bar{P}^c=\langle\bar{Q},\bar{q}_0,\widehat{\bar{\Sigma}}_{proc},\bar{Q}_F,\bar{\delta}\rangle$ *such that*

- $\bar{Q}=Q\uplus\{init,end\}$,

- $\bar{q}_0 = init$,
- $\bar{Q}_F = \{end\}$,
-
  - $\bar{\delta}(q, \hat{\bar{c}}) = \delta(q, \hat{c})\ \forall q \in Q, \hat{\bar{c}} \in \hat{\bar{\Sigma}}_{call}$,
  - $\bar{\delta}(init, \hat{c}) = q_0$,
  - $\bar{\delta}(q, \hat{r}) = end\ \forall q \in Q_F$,
  - $\bar{\delta}(q, \hat{a}) = \delta(q, \hat{a})\ \forall q \in Q, \hat{a} \in \hat{\Sigma}_{int}$.

Note that the above definition of $\bar{\delta}$ is partial. The instrumented procedural automaton can be easily made total (and minimal) via a post-processing step as discussed in Definition 9.

Definition 41 (Instrumented SPA)
Let Σ be an SPA input alphabet with $\Sigma_{call} = \{c_1, \ldots, c_n\}$ and $S = \langle P^{c_1}, \ldots, P^{c_n} \rangle$ be an SPA over Σ. We define the instrumented SPA *over $\bar{\Sigma}$ as $\bar{S} = \langle \bar{P}^{c_1}, \ldots, \bar{P}^{c_n} \rangle$. The* initial procedure *of $\bar{S}$ is denoted as $\bar{c}_0 \in \bar{\Sigma}_{call}$.*

To give an intuition for instrumented SPAs, Figure 4.1 shows the structure of the instrumented SPA based on Figure 3.1. However, note that Figure 4.1 displays a context-free process system (CFPS) and therefore uses a slightly different markup, e.g., the $\bar{c}$-transitions are shown as dashed (name-based) edges with regular c labels, et cetera.

In the following, we look at the comparison between the expansion-SOS of the instrumented SPA and the language-SOS of the original SPA. This requires some utility lemmas first.

Lemma 1
Let Σ be an SPA input alphabet and S denote an SPA over Σ. Let $\hat{u} \in \hat{\Sigma}^*_{proc}, \hat{v} \in \hat{\Sigma}^*, w \in WM(\Sigma)$ and $\sigma \in ST(\Gamma_{SPA})$. We have

$$(\hat{u} \cdot \hat{v}, \sigma)_{SPA} \xrightarrow{w}{}^{*} (\hat{v}, \sigma)_{SPA} \Rightarrow \alpha(w) = \hat{u}.$$

Proof. This is a direct generalization of Lemma 2 of [61]. □

Lemma 2
Let Σ be an SPA input alphabet and P^c be a procedural automaton for some $c \in \Sigma_{call}$. We have

$$L(\bar{P}^c) = \{\hat{\bar{c}}\} \cdot L(P^c)_{\hat{\Sigma}_{call} \mapsto \hat{\bar{\Sigma}}_{call}} \cdot \{\hat{r}\}.$$

Proof. This is a direct consequence of the construction of $\bar{P}^c$ (cf. Definition 40). Here, $L_{X \mapsto \bar{X}}$ represents the language L where for all words $w \in L$, every occurrence of $x \in X$ in w has been replaced with $\bar{x} \in \bar{X}$. □

Lemma 3
Let Σ be an SPA input alphabet and S be an SPA over Σ. Let $\bar{S}$ be the instrumented SPA over $\bar{\Sigma}$ and $c \in \Sigma_{call}, w \in WM(\Sigma)$. We have

$$(\hat{u} \cdot \hat{v}, \sigma)_{SPA,S} \xrightarrow{w}{}^{*} (\hat{v}, \sigma)_{SPA,S} \Leftrightarrow (\hat{u}_{\hat{\Sigma}_{call} \mapsto \hat{\bar{\Sigma}}_{call}} \cdot \hat{v}_{\hat{\Sigma}_{call} \mapsto \hat{\bar{\Sigma}}_{call}}, \bot)_{Exp,\bar{S}} \xrightarrow{w}{}^{*} (\hat{v}_{\hat{\Sigma}_{call} \mapsto \hat{\bar{\Sigma}}_{call}}, \bot)_{Exp,\bar{S}}$$

for some $\hat{u} \in \hat{\Sigma}^*_{proc}, \hat{v} \in \hat{\Sigma}^*, \sigma \in ST(\Gamma_{SPA})$. We use "SPA, S" and "Exp, $\bar{S}$" to emphasize the SPAs that the respective SOS systems refer to.

Proof. "⇒": Via structural induction over w.

- Let $w \in \Sigma_{int}^*$: Internal symbols are only emitted by int-rules (cf. Definition 29) which directly concludes that $\widehat{u} = w \in \Sigma_{int}^*$. Since the instrumentation does not impact internal symbols, we have $\widehat{u}_{\widehat{\Sigma}_{call} \mapsto \widehat{\bar{\Sigma}}_{call}} = \widehat{u}$ and by subsequent application of emission-rules (cf. Definition 38) the statement directly follows.
- Let $w = c \cdot x \cdot r$ with $c \in \Sigma_{call}, x \in WM(\Sigma)$ such that the statement holds for x: By Lemma 1, we can conclude $\widehat{u} = \alpha(w) = \widehat{c}$. We can unfold the premise of the implication to

$$(\widehat{c} \cdot \widehat{v}, \sigma)_{SPA,S} \xrightarrow{c} (\widehat{y} \cdot \widehat{r}, \widehat{v} \bullet \sigma)_{SPA,S} \xrightarrow{x}{}^{*} (\widehat{r}, \widehat{v} \bullet \sigma)_{SPA,S} \xrightarrow{r} (\widehat{v}, \sigma)_{SPA,S}$$

  for some $\widehat{y} \in L(P^c)$. Since $\widehat{u} = \widehat{c}$, we have $\widehat{u}_{\widehat{\Sigma}_{call} \mapsto \widehat{\bar{\Sigma}}_{call}} = \widehat{\bar{c}}$ and by $\widehat{y} \in L(P^c)$ and Lemma 2 we know that $\widehat{c} \cdot \widehat{y}_{\widehat{\Sigma}_{call} \mapsto \widehat{\bar{\Sigma}}_{call}} \cdot \widehat{r} \in L(\bar{P}^c)$. By the expansion-rule of Definition 38 and our induction hypothesis we can then conclude

$$\begin{aligned}(\widehat{\bar{c}} \cdot \widehat{v}_{\widehat{\Sigma}_{call} \mapsto \widehat{\bar{\Sigma}}_{call}}, \bot)_{Exp,\bar{S}} &\xrightarrow{\varepsilon} (\widehat{c} \cdot \widehat{y}_{\widehat{\Sigma}_{call} \mapsto \widehat{\bar{\Sigma}}_{call}} \cdot \widehat{r} \cdot \widehat{v}_{\widehat{\Sigma}_{call} \mapsto \widehat{\bar{\Sigma}}_{call}}, \bot)_{Exp,\bar{S}} \\ &\xrightarrow{c} (\widehat{y}_{\widehat{\Sigma}_{call} \mapsto \widehat{\bar{\Sigma}}_{call}} \cdot \widehat{r} \cdot \widehat{v}_{\widehat{\Sigma}_{call} \mapsto \widehat{\bar{\Sigma}}_{call}}, \bot)_{Exp,\bar{S}} \\ &\xrightarrow{x}{}^{*} (\widehat{r} \cdot \widehat{v}_{\widehat{\Sigma}_{call} \mapsto \widehat{\bar{\Sigma}}_{call}}, \bot)_{Exp,\bar{S}} \\ &\xrightarrow{r} (\widehat{v}_{\widehat{\Sigma}_{call} \mapsto \widehat{\bar{\Sigma}}_{call}}, \bot)_{Exp,\bar{S}}.\end{aligned}$$

- Let $w = w_1 \cdot w_2$ with $w_1, w_2 \in WM(\Sigma)$ such that the statement holds for w_1, w_2: Again, by Lemma 1, we can conclude $\widehat{u} = \alpha(w_1 \cdot w_2)$. Since both w_1, w_2 are independently well-matched, they do not share matching call symbols or return symbols. As a result α can be applied in a piecewise fashion and we have $\widehat{u} = \alpha(w_1 \cdot w_2) = \alpha(w_1) \cdot \alpha(w_2)$. By subsequent application of the induction hypothesis (first with $\widehat{u} = \alpha(w_1)$ and then with $\widehat{u} = \alpha(w_2)$) we can directly conclude

$$\begin{aligned}(\alpha(w_1)_{\widehat{\Sigma}_{call} \mapsto \widehat{\bar{\Sigma}}_{call}} \cdot \alpha(w_2)_{\widehat{\Sigma}_{call} \mapsto \widehat{\bar{\Sigma}}_{call}} \cdot \widehat{v}_{\widehat{\Sigma}_{call} \mapsto \widehat{\bar{\Sigma}}_{call}}, \bot)_{Exp,\bar{S}} & \\ \xrightarrow{w_1}{}^{*} (\alpha(w_2)_{\widehat{\Sigma}_{call} \mapsto \widehat{\bar{\Sigma}}_{call}} \cdot \widehat{v}_{\widehat{\Sigma}_{call} \mapsto \widehat{\bar{\Sigma}}_{call}}, \bot)_{Exp,\bar{S}} & \\ \xrightarrow{w_2}{}^{*} (\widehat{v}_{\widehat{\Sigma}_{call} \mapsto \widehat{\bar{\Sigma}}_{call}}, \bot)_{Exp,\bar{S}} &\end{aligned}$$

  by applying the argumentation from the previous two induction steps depending on whether the expansion-SOS processes call symbols or internal symbols of $\alpha(w_1)$ and $\alpha(w_2)$.

"⇐": Analogously to the first direction by arguing each case in reverse order. □

Theorem 3
Let Σ be an SPA input alphabet and S denote an SPA over Σ. Let $\bar{S}$ be the instrumented SPA

over $\bar{\Sigma}$. *Then we have*

$$w \in L(S) \Leftrightarrow w \in Exp(\bar{S})$$

for all $w \in \Sigma^*$.

Proof. We use "SPA, S" and "$Exp, \bar{S}$" to emphasize the SPAs that the respective SOS systems refer to. We have

$$w \in L(S) \Leftrightarrow (\widehat{c}_0, \bot)_{SPA,S} \xrightarrow{w}{}^*(\varepsilon, \bot)_{SPA,S} \tag{3.1}$$

$$\Leftrightarrow (\widehat{c}_0 \cdot \varepsilon, \bot)_{SPA,S} \xrightarrow{w}{}^*(\varepsilon, \bot)_{SPA,S} \tag{3.2}$$

$$\Leftrightarrow (\widehat{\widehat{c}}_0 \cdot \varepsilon, \bot)_{Exp,\bar{S}} \xrightarrow{w}{}^*(\varepsilon, \bot)_{Exp,\bar{S}} \tag{3.3}$$

$$\Leftrightarrow (\widehat{\widehat{c}}_0, \bot)_{Exp,\bar{S}} \xrightarrow{w}{}^*(\varepsilon, \bot)_{Exp,\bar{S}} \tag{3.4}$$

$$\Leftrightarrow w \in Exp(\bar{S}) \tag{3.5}$$

Equation (3.1) directly follows from Definition 29. Equation (3.3) follows from Lemma 3 and Equation (3.5) follows from Definition 38. □

Figure 3.4 summarizes the relationship of the two approaches. The base system may be described by either a CFG or an SPA, as they are only different syntactical representations for the same information. For constructing the (instrumented) language of the base system, one may either use the language-SOS (cf. Definition 29) directly or the two-step process involving instrumentation (cf. Definition 24) and expansion (cf. Definition 38). Theorem 3 shows that both approaches describe the same language, so it is merely a question of convenience which one to choose.

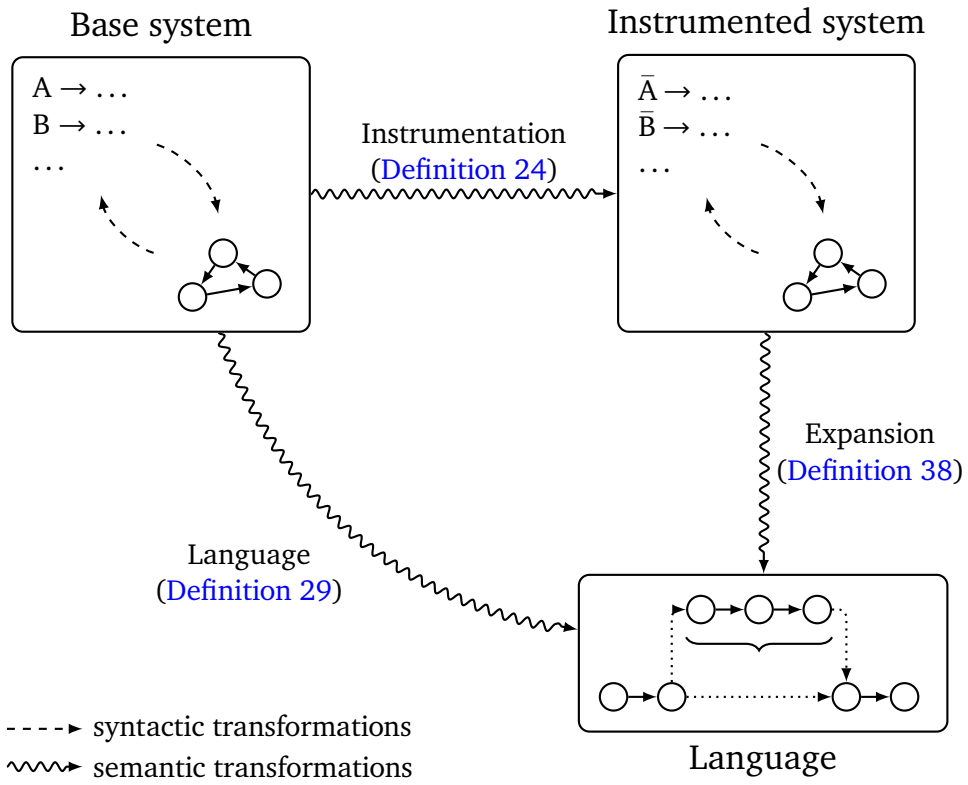
3.3 SBAs

As motivated in Chapter 1, a central requirement for the successful application of MBQA in practice is the ability to precisely describe system behavior with models. SPAs describe a system holistically: The language of an SPA is minimally well-matched so it is necessary to always consider full runs of a system in order to reason about their validity as ill-matched runs are invalid by default. This property may cause challenges in practice because it forces runs of a system to terminate with an empty "stack". For example, if an erroneous behavior is encountered within a nested procedural invocation, the concerned run still needs to correctly terminate in order to not mask this error.

One can lift this constraint by considering *prefix-closed languages* for modeling system behavior, i.e, languages L such that for every $w \in L$, $Pref(w) \subseteq L$ holds. For example, consider the word $F \cdot a \cdot F \cdot G \cdot c \cdot R \cdot R \cdot a \cdot R$ from Example 4. To decide whether F, $F \cdot a$, etc. are still valid steps in a system, it would be convenient to check whether F, $F \cdot a$, etc. are members of the language of the SPA.

However, introducing prefix-closure to an SPA language directly impacts the inherent link between the global language of an SPA and the local languages of its procedures (cf. Theorem 1) as this link is built upon the holistic interpretation that procedures describe

Figure 3.4
The relationship between the instrumentation, expansion, and language of SPAs.



terminating invocations. Especially, one cannot simply consider prefix-closed procedural automata. Consider the SPA constructed from the two automata of Figure 3.1. Making procedure F accept the word $a \cdot F$ does not make the induced SPA accept the word $F \cdot a \cdot F$. Instead, the corresponding SPA would *wrongfully* accept $F \cdot a \cdot F \cdot R \cdot R$, i.e., a word that is not a prefix of any $w \in L(G_{palin-inst})$ (cf. Example 3).

The crux of this issue is the question about what knowledge the local, procedural languages encode. In the language-SOS of an SPA, call-rules use procedural words to progress the SOS configurations by setting the concatenation of the procedural word and the return symbol as the new system state. Here, procedural languages encode *termination* because they decide when returning, i.e., emitting the r symbol, is possible. For prefix-closed semantics, it is necessary to encode *reachability*. These are two different properties: In arbitrary DFAs, parsing an accepted word w may at some point traverse non-accepting states. So while w is terminating and consequently reachable, there exist prefixes of w that are not terminating but still reachable.

The challenge for prefix-closed systems is that termination and reachability are two separate concepts that cannot be simultaneously encoded via the binary acceptance criterion of DFAs. A possible way to tackle this issue is to model termination explicitly by including the return symbol in the input alphabet of a procedural automaton and treating it as a first-class citizen of the procedural language. This idea leads to the notion of *behavioral automata* and consequently *SBAs*.

Remark 4
Note that [62] uses the terms "returnability" and "returnable" to describe the concepts of "termination" and "terminating" procedures.

Definition 42 (Behavioral automaton [62])
Let Σ be an SPA input alphabet and $c \in \Sigma_{call}$ denote a procedure. A behavioral automaton *for procedure c is a DFA $P_B^c = \langle Q^c, q_0^c, \widehat{\Sigma}, Q_F^c, \delta^c \rangle$.*

Definition 43 (SBA [62])
Let Σ be an SPA input alphabet with $\Sigma_{call} = \{c_1, \ldots, c_n\}$. An SBA over Σ is a tuple $S_B = \langle P_B^{c_1}, \ldots, P_B^{c_n} \rangle$ such that for each call symbol there exists a corresponding behavioral automaton. The initial procedure *of S_B is denoted as $c_0 \in \Sigma_{call}$.*

By including the return symbol in the input alphabet of behavioral automata, termination is no longer managed externally, e.g., via a language-SOS, but internally via explicit transitions. However, this change requires additional constraints on the local languages in order to guarantee a semantically correct description of valid system behavior.

Definition 44 (Validity of SBAs)
Let Σ be an SPA input alphabet and S_B be an SBA over Σ. Let Term $\subseteq \Sigma_{call}$ denote the set of terminating procedures. *A procedure c is terminating iff*

$$\exists \widehat{w} \in L(P_B^c) \colon \widehat{w}[|\widehat{w}|] = \widehat{r}.$$

We call S_B valid, *if each behavioral automaton P_B^c is valid. A behavioral automaton P_B^c is valid iff it satisfies the following properties:*

- *prefix-closure, i.e,*

$$\widehat{w} \in L(P_B^c) \Rightarrow \widehat{v} \in L(P_B^c) \quad \forall \widehat{w} \in \widehat{\Sigma}^*, \widehat{v} \in Pref(\widehat{w}).$$

- *return-closure, i.e,*

$$\widehat{w} \in L(P_B^c) \Rightarrow \widehat{w} \cdot \widehat{r} \cdot \widehat{v} \notin L(P_B^c) \quad \forall \widehat{w} \in \widehat{\Sigma}^*, \widehat{v} \in \widehat{\Sigma}^+.$$

- *call-closure, i.e.,*

$$\widehat{w} \in L(P_B^c) \Rightarrow \widehat{w} \cdot \widehat{p} \cdot \widehat{v} \notin L(P_B^c) \quad \forall \widehat{w} \in \widehat{\Sigma}^*, \widehat{p} \notin Term, \widehat{v} \in \widehat{\Sigma}^+.$$

Intuitively, *prefix-closure* allows one to check on a language level whether a run traverses reachable transitions. Once a system diverges from such a path, i.e, a word is rejected, every continuation of this word is also rejected. *Return-closure* ensures that behavioral automata are procedurally consistent. Since the return symbol denotes the end of a procedural invocation, any behavior beyond the first occurrence of $\widehat{r}$ should generally be inaccessible as the (global) behavior returns to the procedure that called the respective behavioral automaton. Similarly, *call-closure* enforces that continuations of non-terminating procedures should not represent reachable behavior on a procedural level. In the following, we only consider valid SBAs unless specified otherwise.

3.3.1 Semantics

For defining the language of SBAs, we use SOS again. For modelling the control component of an SOS configuration, we re-use the stack domain of SPAs (cf. Definition 28).

Definition 45 (Language-SOS of an SBA [62])
Let Σ be an SPA input alphabet and S_B be an SBA over Σ. Using tuples from $\widehat{\Sigma}^* \times ST(\Gamma_{SPA})$ to denote a system configuration, we define three kinds of SOS transformation rules:

1. *call-rules:*

$$\frac{\widehat{w} \in L(P_B^c)}{(\widehat{c} \cdot \widehat{v}, \sigma)_{SBA} \xrightarrow{c} (\widehat{w}, \widehat{v} \bullet \sigma)_{SBA}}$$

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

2. *int-rules:*

$$\frac{-}{(\widehat{a} \cdot \widehat{v}, \sigma)_{SBA} \xrightarrow{a} (\widehat{v}, \sigma)_{SBA}}$$

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

3. *ret-rules:*

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

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

The language *of an SBA S_B is defined as*

$$L(S_B) = \{w \in \Sigma^* \mid \exists \sigma \in ST(\Gamma_{SPA}) \colon (\widehat{c}_0, \bot)_{SBA} \xrightarrow{w}{}^*(\varepsilon, \sigma)_{SBA}\}.$$

We call a word $w \in \Sigma^*$ admissible *in S_B iff $\exists \widehat{v} \in \widehat{\Sigma}^*, \sigma \in ST(\Gamma_{SPA})$ such that*

$$(\widehat{c}_0, \bot)_{SBA} \xrightarrow{w}{}^*(\widehat{v}, \sigma)_{SBA}.$$

There exist two fundamental differences between the definitions of SPA languages and SBA languages. First, termination is now encoded in the languages of behavioral automata as they range over the complete input alphabet $\widehat{\Sigma}$. As a result, call-rules no longer enforce return symbols when extending the SOS state but delegate this decision to words of the involved procedural automata. The constraints on valid SBAs (cf. Definition 44) ensure that this change still results in valid global behavior by including call symbols to non-terminating procedures or the return symbol at most once as the last symbol. Second, the language definition no longer requires an empty stack for termination. While Definition 45 still allows for minimally well-matched words, the language may now also contain minimally return-matched words.

These changes make SBA languages prefix-closed. As all SOS rules emit the first symbol of the current state,

$$(\widehat{c}_0, \bot)_{SBA} \xrightarrow{w}{}^*(\varepsilon, \sigma_1)_{SBA}$$

for some $\sigma_1 \in ST(\Gamma_{SPA})$ directly implies

$$(\widehat{c}_0, \bot)_{SBA} \xrightarrow{u}{}^*(\widehat{v}, \sigma_2)_{SBA} \xrightarrow{v} (\varepsilon, \sigma_1)_{SBA}$$

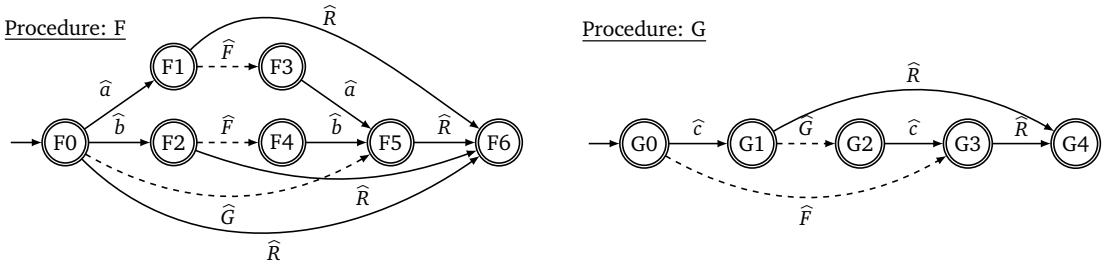
for some $\sigma_1, \sigma_2 \in ST(\Gamma_{SPA})$ and $w = u \cdot v$. Due to the prefix-closure of behavioral automata, one is always able to find a procedural word that omits the symbol $\widehat{v}$ so that the SOS system terminates on $(\varepsilon, \sigma_2)_{SBA}$ after emitting u. Via induction, one can then directly conclude that for any $w \in L(S_B)$, there exists $w' \in L(S_B)$ for all $w' \in Pref(w)$ as well.

It should be noted that SBAs are able to describe even more languages than just prefix-closed SPA languages. Due to the possibility to accept words with arbitrary stack contents, SBAs can represent systems with *non-terminating* procedures. This may be especially useful in cases of live systems which exhibit some form of non-terminating main-loop. Here, there exists no suitable SPA-based representation which would require the termination of all procedures. Instead, SBAs allow for such representations that go beyond the prefix-closure of SPA systems.

To give an intuition of the semantics of SBAs, let us consider a variation of Example 4 in which a prefix of the originally well-matched word is emitted. In order to reason about the traversal of an SBA-based SOS system, we first need to transform the involved procedures to behavioral automata. This involves incorporating the return symbol and adjusting the acceptance of states accordingly. Figure 3.5 shows the transformed behavioral automata and Example 6 shows an exemplary run of the SOS system of the respective SBA.

Example 6 (A run of the SBA based on Figure 3.5)
The SOS system starts with the configuration $(\widehat{F}, \bot)_{SBA}$. Since $\widehat{a} \cdot \widehat{F} \cdot \widehat{a} \in L(P_B^F)$, we can apply

Figure 3.5 (from [62])
Behavioral automata based on the automata of Figure 3.1. Sink states and corresponding transitions are omitted for readability.



a call-rule to perform the transition

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

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

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

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

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

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

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

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

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

Now we use a ret-rule to parse the return symbol

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

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

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

Since $\varepsilon \bullet \widehat{a} \bullet \varepsilon \bullet \bot$ is a valid element of $ST(\Gamma_{SPA})$, we have $F \cdot a \cdot F \cdot G \cdot c \cdot R \in L(S_B)$.

Recall that Definition 45 does not require an empty stack to emit a word and especially not a "canonical" stack in the form of $\varepsilon \bullet \varepsilon \bullet \ldots \bullet \bot$. While the prefix-closure of behavioral

automata would certainly allow for such stack configurations, it would be an unnecessary constraint as reaching the empty state ε is the determining factor for the termination.

Similar to SPAs, we continue with a series of properties of SBAs.

Definition 46 (Properties of SBAs)
Let Σ be an SPA input alphabet and $S_B = \langle P_B^c, \ldots \rangle, S_{B1}, S_{B2}$ denote some SBAs over Σ.

- *We write $|S_B| = \sum_{c \in \Sigma_{call}} |P_B^c|$ to denote the* size *of an SBA.*
- *We call an SBA S_B* minimal *(with respect to Σ) iff*
  - *each behavioral automaton P_B^c is minimal (with respect to $\widehat{\Sigma}$) and*
  - *$\forall c \in \Sigma_{call} \colon \exists w \in L(S_B), i \in \{1, \ldots, |w|\} \colon w[i] = c$.*

  Note that the latter property requires all behavioral automata to be reachable (by means of the language-SOS system) and describe a non-empty language.
- *We call an SBA S_{B1}* equivalent *to another SBA S_{B2} (denoted as $S_{B1} \equiv_{SBA} S_{B2}$) iff $L(S_{B1}) = L(S_{B2})$.*

In the following, we only consider minimal SBAs with respect to Σ unless specified otherwise. For SBAs, minimality also implies canonicity because minimal DFAs are unique (up to isomorphism) and SBAs are uniquely defined by their (canonical) behavioral automata.

3.3.2 (De-) Composition Properties

In analogy to SPAs, SBAs pursue the same idea of translating between the global, instrumented language and local, procedural languages of the respective behavioral automata via projection and expansion. However, by adding the return symbol to the input alphabet of behavioral automata, termination, i.e., the emission of the return symbol r, is now a procedural decision. Furthermore, SBA languages are prefix-closed. These two changes impact the ability to translate between the two views.

In the following, we look at the necessary adjustments for the projection and expansion of SBA words. We continue with an SBA-specific adaption of Lemma 1 which leads to the corresponding theorems for SBAs afterwards.

Lemma 4
Let Σ be an SPA input alphabet, $\widehat{u} \in \widehat{\Sigma}^*_{proc}, \widehat{v} \in \widehat{\Sigma}^*, w \in WM(\Sigma)$ and $\sigma \in ST(\Gamma_{SPA})$. We have

$$(\widehat{u} \cdot \widehat{v}, \sigma)_{SBA} \xrightarrow{w}{}^{*} (\widehat{v}, \sigma)_{SBA} \Rightarrow \alpha(w) = \widehat{u}.$$

Proof. Similar to Lemma 1, this is a direct generalization of Lemma 2 of [61]. Note that while we discuss the language-SOS of SBAs, the constraint of $w \in WM(\Sigma)$ forces all (potentially nested) calls occurring in w to have a matching return symbol in w as well. As a result, we can apply the argumentation of the well-matched SPA case without problems. □

Theorem 4 (Behavioral projection theorem [62])
Let Σ be an SPA input alphabet and S_B be an SBA over Σ. Let $w \in MRM(\Sigma)$ be a non-empty,

minimally return-matched word starting with c_0. Then we have

$$w \in L(S_B) \Leftrightarrow \forall \langle c, i\rangle \in Inst_w \colon \alpha(u) \cdot \widehat{v} \in L(P_B^c)$$

where $u = w[i+1, j]$, $j = \rho_w(i+1)$ *and* $\widehat{v} = \begin{cases} \varepsilon & \text{if } j = |w| \\ \widehat{w}[j+1] & \text{otherwise} \end{cases}$

Proof. "⇒": Let $w \in L(S_B)$ and $\langle c, i\rangle \in Inst_w$ be arbitrary. We distinguish between the two cases for $\widehat{v}$.

- Let $\widehat{v} = \varepsilon$. Since $w[i]$ is a call symbol and $w[i+1, j]$ is a well-matched word of maximum length, we know by Definition 45 that

  $$(\widehat{c} \cdot \widehat{x}, \sigma)_{SBA} \xrightarrow{c} (\widehat{y}, \widehat{x} \bullet \sigma)_{SBA} \xrightarrow{w[i+1,j]}{}^{*} (\varepsilon, \widehat{x} \bullet \sigma)_{SBA}$$

  for some $\widehat{x} \in \widehat{\Sigma}^*, \widehat{y} \in \widehat{\Sigma}^*_{proc}, \sigma \in ST(\Gamma_{SPA})$. Note that $\widehat{y}$ cannot contain an $\widehat{r}$ symbol because $w[i+1, j]$ is well-matched and $j = |w|$. By Lemma 4, we know that $\alpha(w[i+1, j]) = \widehat{y}$. Due to the applicability of the call-rule, we know that $\widehat{y} \in L(P_B^c)$ and the statements directly follows.

- Let $\widehat{v} = \widehat{w}[j+1]$. We know that $w[j+1]$ is either a call symbol or the return symbol because otherwise j would not be the maximum index such that $w[i+1, j] \in WM(\Sigma)$, contradicting our assumption. By Definition 45, call-rules and ret-rules (for emitting such a symbol) require the first symbol of the current SOS configuration to be a call symbol or the return symbol as well. We have

  $$(\widehat{c} \cdot \widehat{x}, \sigma_1)_{SBA} \xrightarrow{c} (\widehat{y}_1 \cdot \widehat{v} \cdot \widehat{y}_2, \sigma_2)_{SBA} \xrightarrow{w[i+1,j]}{}^{*} (\widehat{v} \cdot \widehat{y}_2, \sigma_2)_{SBA} \xrightarrow{v} (\widehat{z}, \sigma_3)_{SBA}$$

  for some $\widehat{x}, \widehat{y}_2, \widehat{z} \in \widehat{\Sigma}^*, \widehat{y}_1 \in \widehat{\Sigma}^*_{proc}, \sigma_1, \sigma_2, \sigma_3 \in ST(\Gamma_{SPA})$. Similar to the previous case, we know by Lemma 4 that $\alpha(w[i+1, j]) = \widehat{y}_1$ and according to the call-rules of the language-SOS $\widehat{y}_1 \cdot \widehat{v} \cdot \widehat{y}_2 \in L(P_B^c)$. Due to the prefix-closure of P_B^c, we have $\widehat{y}_1 \cdot \widehat{v} \in L(P_B^c)$ and the statement directly follows.

"⇐": We show via contraposition that

$$w \notin L(S_B) \Rightarrow \exists \langle c, i\rangle \in Inst_w \colon \alpha(u) \cdot \widehat{v} \notin L(P_B^c).$$

Without loss of generality, let $w = w_1 \cdot w_2$ for some $w_1 \in \Sigma^*, w_2 \in \Sigma$. Due to the prefix-closure of $L(S_B)$, we can shorten every rejected word to a word that is rejected because of its last symbol, i.e., there exist $\widehat{x}_1 \in \widehat{\Sigma}^*, \sigma_1 \in ST(\Gamma_{SPA})$ such that

$$(\widehat{c}_0, \bot)_{SBA} \xrightarrow{w_1}{}^{*} (\widehat{x}_1, \sigma_1)_{SBA}$$

but $\nexists\, \widehat{x}_2 \in \Sigma^*, \sigma_2 \in ST(\Gamma_{SPA})$ such that

$$(\widehat{c}_0, \bot)_{SBA} \xrightarrow{w_1 \cdot w_2}{}^{*} (\widehat{x}_2, \sigma_2)_{SBA}.$$

Such a decomposition has to exist because otherwise (i.e., $w_1 \cdot w_2 = w$, $\widehat{x}_2 = \varepsilon$, $\sigma_2 \in ST(\Gamma_{SPA})$) w would be in the language of the SBA, contradicting our assumption. Let $\langle c_*, i_* \rangle \in Inst_w$ such that $i_* \leq |w_1|$ is the largest possible index. This means, it is the context of procedure c_* in which the symbol w_2 cannot be emitted. We distinguish whether w_2 is a call symbol, an internal symbol, or the return symbol.

- Let $w_2 \in \Sigma_{call}$. Then w_2 cannot be an element of the maximum well-matched word $w[i_* + 1, j]$ and therefore $\widehat{v} = \widehat{w}_2$. We have

$$(\widehat{c}_* \cdot \widehat{x}, \sigma_1)_{SBA} \xrightarrow{c_*} (\widehat{y} \cdot \widehat{w}_2, \sigma_2)_{SBA} \xrightarrow{w[i_*+1,j]}{}^{*} (\widehat{w}_2, \sigma_2)_{SBA} \xrightarrow{w_2} (\widehat{z}, \sigma_3)_{SBA}$$

  for some $\widehat{x}, \widehat{z} \in \widehat{\Sigma}^*, \widehat{y} \in \widehat{\Sigma}^*_{proc}, \sigma_1, \sigma_2, \sigma_3 \in ST(\Gamma_{SPA})$. Note that we do not need to consider any continuations beyond $\widehat{w}_2$ because w is shortened. By Lemma 4, we know that $\alpha(w[i_* + 1, j]) = \widehat{y}$. Since the last transformation is not applicable, we can directly conclude from the call-rule of c_* that $\alpha(w[i_* + 1, j]) \cdot \widehat{v} \notin L(P_B^{c_*})$.

- Let $w_2 \in \Sigma_{int}$. Then w_2 is the last symbol of the maximum well-matched word $w[i_* + 1, j]$ and therefore $\widehat{v} = \varepsilon$. We have

$$(\widehat{c}_* \cdot \widehat{x}, \sigma_1)_{SBA} \xrightarrow{c_*} (\widehat{y} \cdot \widehat{w}_2, \sigma_2)_{SBA} \xrightarrow{w[i_*+1,j-1]}{}^{*} (\widehat{w}_2, \sigma_2)_{SBA} \xrightarrow{w_2} (\widehat{z}, \sigma_3)_{SBA}$$

  for some $\widehat{x}, \widehat{z} \in \widehat{\Sigma}^*, \widehat{y} \in \widehat{\Sigma}^*_{proc}, \sigma_1, \sigma_2, \sigma_3 \in ST(\Gamma_{SPA})$. Note that we do not need to consider any continuations beyond $\widehat{w}_2$ because w is shortened. Since $w[i_*+1, j-1]$ is also well-matched, we can conclude by Lemma 4 that $\alpha(w[i_* + 1, j - 1]) = \widehat{y}$ and since α is applied piecewise for internal symbols, we can conclude $\alpha(w[i_* + 1, j]) = \widehat{y} \cdot \widehat{w}_2$. Since the last transformation is not applicable, we can directly conclude from the call-rule of c_* that $\alpha(w[i_* + 1, j]) \notin L(P_B^{c_*})$.

- Let $w_2 = r$. The argumentation is identical to the case of $w_2 \in \Sigma_{call}$. □

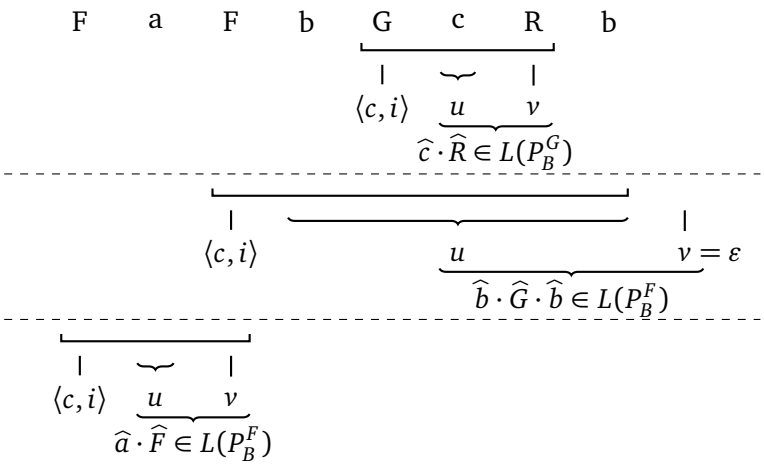
To give an intuition for this projection, Figure 3.6 shows an example of the three different cases for $\widehat{v}$ ($\widehat{v} = \varepsilon, \widehat{v} \in \widehat{\Sigma}_{call}, \widehat{v} = \widehat{r}$) for a word of the SBA based on Figure 3.5.

Similar to SPAs, the *expansion process* of local words of behavioral automata requires a set of sequences for transforming them into words for the global, instrumented system. However, contrary to SPAs, the prefix-closure of SBA languages allows one to omit return sequences for this process. We continue with the SBA-specific formalization of access sequences and terminating sequences and the respective expansion process afterwards.

Definition 47 (Access sequences and terminating sequences for SBAs)
Let Σ be an SPA input alphabet and S_B be an SBA over Σ. The set of access sequences *of procedure $c \in \Sigma_{call}$ (denoted as $AS_c \subseteq \Sigma^*$) and the set of* terminating sequences *of procedure $c \in \Sigma_{call}$ (denoted as $TS_c \subseteq \Sigma^*$) are defined as*

$$AS_c = \{w \in L(S_B) \mid w[|w|] = c\} \text{ and}$$
$$TS_c = \{w \in WM(\Sigma) \mid \exists as \in AS_c : as \cdot w \cdot r \in L(S_B)\}.$$

Figure 3.6 (from [62])
A visualization of the different (de-) composition cases of Theorem 4 for a word of the SBA based on Figure 3.5.



Note that the separate definition of AS_c and TS_c (compared to, e.g., Definition 35) is necessary to account for the possibility of non-terminating procedures, i.e., procedures which have access sequences but no terminating sequences. Furthermore, due to the prefix-closure of SBAs and the omission of the return sequence, it is necessary to append an additional return symbol in order to verify the correctness of the terminating sequences. Again, as_c and ts_c may be used as a shorthand notation for elements $as \in AS_c$ and $ts \in TS_c$.

Theorem 5 (Behavioral expansion theorem)
Let Σ be an SPA input alphabet and $S_B = \langle P_B^c, \ldots \rangle$ be an SBA over Σ. Let $\widehat{w} = \widehat{u} \cdot \widehat{v}$ with $\widehat{u} \in \widehat{\Sigma}^*_{proc}, \widehat{v} \in \widehat{\Sigma}$. We have

$$\widehat{u} \cdot \widehat{v} \in L(P_B^c) \Longleftrightarrow as \cdot \gamma(\widehat{u}) \cdot v \in L(S_B)$$

for all $c \in \Sigma_{call}, as \in AS_c$.
Note that due to the return-closure and call-closure of valid behavioral automata, any word that extends beyond a non-continuable symbol is rejected by default and therefore cannot be used in the states of the language-SOS system of SBAs. As a result, only $\widehat{v}$ can be non-continuable, $\widehat{u}$ can be constrained to $\widehat{\Sigma}^*_{proc}$, and γ can expand every call symbol occurring in $\widehat{u}$. Furthermore, the above expansion does not account for the empty word. Since valid behavioral automata are prefix-closed by construction and the empty word is the trivial element of any (non-empty) prefix-closed language, there is no practical need to explicitly cover this case.

Proof. Let $c \in \Sigma_{call}$ and $as \in AS_c$ be arbitrary.
"$\Leftarrow$": By Definitions 45 and 47, we know from the premise of the implication that

$$(\widehat{c}_0, \bot)_{SBA} \xrightarrow{as}{}^{*} (\widehat{x}, \sigma_1)_{SBA} \xrightarrow{\gamma(\widehat{u}) \cdot v}{}^{*} (\varepsilon, \sigma_2)_{SBA}$$

for some $\widehat{x} \in L(P_B^c), \sigma_1, \sigma_2 \in ST(\Gamma_{SPA})$. Since the emission of a single symbol v requires the consumption of a single symbol from the state of the SOS configuration, we can further restrict $\widehat{x}$ to be of the form $\widehat{x} = \widehat{y} \cdot \widehat{z}$ with $\widehat{y} \in \widehat{\Sigma}^*_{proc}, \widehat{z} \in \widehat{\Sigma}$, i.e,

$$(\widehat{c}_0, \bot)_{SBA} \xrightarrow{as}{}^*(\widehat{y} \cdot \widehat{z}, \sigma_1)_{SBA} \xrightarrow{\gamma(\widehat{u})}{}^*(\widehat{z}, \sigma_1)_{SBA} \xrightarrow{v} (\varepsilon, \sigma_2)_{SBA}.$$

What remains to be shown is that $\widehat{y} \cdot \widehat{z} = \widehat{u} \cdot \widehat{v}$. Since γ always emits well-matched words, we can apply Lemma 4 and know that $\alpha(\gamma(\widehat{u})) = \widehat{y}$. Since α is the inverse function of γ, we can directly conclude that $\widehat{y} = \alpha(\gamma(\widehat{u})) = \widehat{u}$. Since the emission of a single symbol v requires the first symbol of the state of the SOS configuration to be $\widehat{v}$, we can directly conclude that $\widehat{z} = \widehat{v}$. This concludes the statement.
"⇒": We show via contraposition that $as \cdot \gamma(\widehat{u}) \cdot v \notin L(S_B) \Rightarrow \widehat{u} \cdot \widehat{v} \notin L(P_B^c)$. Similar to the argumentation of the "⇐"-case of Theorem 4, we know that $as \cdot \gamma(\widehat{u}) \cdot v \notin L(S_B)$ implies a non-applicable transformation in the SOS system, i.e.,

$$(\widehat{c}_0, \bot)_{SBA} \xrightarrow{as}{}^*(\widehat{x}, \sigma_1)_{SBA} \xrightarrow{\gamma(\widehat{u})\cdot v}{}^*(\varepsilon, \sigma_2)_{SBA}$$

for some $\widehat{x} \in \widehat{\Sigma}^*$. Assume that $\widehat{x} \in L(P_B^c)$. Using the same argumentation from the "⇐"-case of this theorem, we can conclude that $\widehat{x} = \alpha(\gamma(\widehat{u})) \cdot \widehat{v} = \widehat{u} \cdot \widehat{v}$. However, in this case we have

$$(\widehat{x}, \sigma_1)_{SBA} \xrightarrow{\gamma(\widehat{u})\cdot v}{}^*(\varepsilon, \sigma_2)_{SBA},$$

contradicting our assumption. As a result, we can conclude $\widehat{x} = \widehat{u} \cdot \widehat{v} \notin L(P_B^c)$. □

Intuitively, Theorem 5 establishes a similar expansion mechanism to Theorem 2 but with the exclusion of the last symbol of each local word. Due to the return-closure and call-closure of valid behavioral automata, any word that continues beyond a call (symbol) to a non-terminating procedure or the return symbol is rejected by default and therefore does not require expansion. Only for words where these symbols do not occur at all or as the last symbol, querying the global system is relevant. Consequently, if expansion is required, $\widehat{u}$ only contains internal symbols or call symbols for which terminating sequences exist and therefore can be correctly processed by γ.

Similar to SPAs, Theorem 4 provides an alternative characterization of SBA equivalence.

Corollary 2 (Equivalence of SBAs)
Let Σ be an SPA input alphabet and $S_{B1} = \langle P_{B1}^c, \ldots \rangle, S_{B2} = \langle P_{B2}^c, \ldots \rangle$ be two SBAs over Σ. We have

$$S_{B1} \equiv_{SBA} S_{B2} \iff \forall c \in \Sigma_{call} \colon P_{B1}^c \equiv_{DFA} P_{B2}^c.$$

Proof. The argumentation is identical to Corollary 1 using the projection of Theorem 4. □

3.3.3 Reductions

SBAs extend the concept of SPAs by introducing the property of prefix-closure to their languages. For any given SPA S, one can construct a corresponding SBA S_B such that

$L(S_B) = \{w \mid w \in Pref(w'), w' \in L(S)\}$. Internally, this is achieved by distinguishing between termination and reachability via prefix-closed behavioral automata that explicitly model return transitions. In the following, we briefly look at the *reverse* process, i.e., *reducing* the language of an SBA to its well-matched core via a transformation back into an SPA.

Reduction is an interesting concept as it describes the intersection of SBA languages with the set of well-matched words $WM(\Sigma)$. Its motivation lies in the (de-) composition properties of SBAs, specifically the expansion of local words to global words. Compared to the expansion process of SPAs, SBA expansion does not require concatenating return sequences to transformed local words, allowing one to skip symbols. Especially in the context of MBT and AAL, this has practical relevance. In MBQA, where words correspond to input sequences on a system, reducing the length of words directly reduces the amount of steps a system has to execute during testing or learning. If the system supports prefix-closed semantics, one may *simulate* SPA-based testing or learning via SBAs and a reduction afterwards. The evaluation in [62] shows that this approach can boost the performance of these processes in some situations. However, note that this process is most of the time only reasonable for SBAs which consist of only terminating (cf. Definition 44) procedures as non-terminating procedures cannot be represented by SPAs. The reduction of a non-terminating procedure would make it unreachable in the SPA-based interpretation.

Formally, reducing an SBA is a straight forward process. Recall from the motivation of behavioral automata that termination needs to be explicitly encoded via return transitions. In order to reverse this process, one only needs to look at the acceptance of return successors to determine the acceptance of states in a procedural automaton.

Definition 48 (Reduction of a behavioral automaton)
Let Σ be an SPA input alphabet and $P_B^c = \langle Q^c, q_0^c, \widehat{\Sigma}, Q_F^c, \delta^c \rangle$ denote a behavioral automaton for a procedure $c \in \Sigma_{call}$. We define the reduced behavioral automaton *$R(P_B^c) = \langle R(Q^c), R(q_0^c), \widehat{\Sigma}_{proc}, R(Q_F^c), R(\delta^c) \rangle$ as follows:*

- $R(Q^c) = Q^c$,
- $R(q_0^c) = q_0^c$,
- $R(\delta^c) = \{\langle p, \widehat{a}, q \rangle \in \delta^c | \widehat{a} \in \widehat{\Sigma}_{proc}\}$, *and*
- $R(Q_F^c) = \{p \in Q^c | \exists q \in Q_F^c : \delta^c(p, \widehat{r}) = q\}$.

Note that the above construction not necessarily constructs a minimal automaton, as it may contain equivalent and unreachable states. This is only a technical detail, as one can easily minimize arbitrary DFAs (cf. Definition 9).

The concept of reduction is then generalized to an SBA by reducing each of its behavioral automata by means of Definition 48 and constructing an SPA from these reduced automata. To give an intuition for this process, consider the SBA based on Figure 3.5. The *reduced SBA* is described by the SPA based on Figure 3.1.

3.4 SPMMs

With SPAs and SBAs, the previous sections present two intuitive formalisms for describing (prefix-closed) languages of instrumented context-free systems. However, as with generic

formal languages in the context of MBQA in general, they are limited to only distinguish between valid and invalid behavior of a system. Either a word is a member of the respective language, i.e., representing a successful run of the system, or not, i.e., representing a failed run of the system. This section discusses a formalism tailored towards *reactive* systems which offer clients a dialog-based form of interaction consisting of input actions and observable output reactions. As discussed in Section 1.1.2, the focus is on deterministic transductions that follow an incremental lock-step pattern known from Mealy machines in case of regular input languages.

Semantically, these systems are based on SBAs. Recall from Section 2.1.3 that Mealy-based transductions can be easily represented via prefix-closed DFAs. With SBAs, not only does there exist a globally prefix-closed formalism, but also its procedural components, i.e., the behavioral automata, are prefix-closed as well. This directly suggests the idea of systems of procedural Mealy machines (SPMMs). However, as discussed in Section 2.1.3 as well, there exist certain pitfalls when representing Mealy-based transductions with DFAs. Specifically in the context of instrumented systems, the roles of symbols pose additional challenges as, e.g., call symbols or the return symbol need to maintain their roles throughout possible transformations.

We continue with the syntactical introduction of SPMMs and their components and look at the semantic properties afterwards.

Definition 49 (SPA output alphabet)
An SPA output alphabet *is a disjoint union* $\Omega = \Omega_{int} \uplus \{👍, 👎\}$ *where* Ω_{int} *denotes the* internal output alphabet, 👍 *denotes the* "success" output symbol, *and* 👎 *denotes the* "error" output symbol.

Similar to SPA input alphabets, we use $\widehat{\ }$ (as in $\widehat{\Omega}$) to denote output symbols that are interpreted in a local, procedural context and add (remove) this markup token to switch between local and global contexts.

Definition 50 (Procedural Mealy machine)
Let Σ *be an SPA input alphabet,* Ω *be an SPA output alphabet and* $c \in \Sigma_{call}$ *denote a procedure. A* procedural Mealy machine *for procedure* c *over* Σ *and* Ω *is a Mealy machine* $P_M^c = \langle Q^c, q_0^c, \widehat{\Sigma}, \widehat{\Omega}, \delta^c, \lambda^c \rangle$.

Definition 51 (SPMM)
Let Σ *be an SPA input alphabet with* $\Sigma_{call} = \{c_1, \ldots, c_n\}$ *and* Ω *be an SPA output alphabet. An* SPMM *over* Σ *and* Ω *is a tuple* $S_M = \langle P_M^{c_1}, \ldots, P_M^{c_n} \rangle$ *such that for each call symbol there exists a corresponding procedural Mealy machine. The* initial procedure *of* S_M *is denoted as* $c_0 \in \Sigma_{call}$.

In order to formalize the semantics of SPMMs on the basis of SBAs, we need a notion of *validity* similar to Definition 44.

Definition 52 (Validity of SPMMs)
Let Σ *be an SPA input alphabet,* Ω *be an SPA output alphabet, and* S_M *be an SPMM over* Σ *and*

Ω. *Let* $Term \subseteq \Sigma_{call}$ *denote the set of* terminating procedures. *A procedure* c *is terminating iff*

$$\exists q \in Q^c : \lambda^c(q, \widehat{r}) = \widehat{👍}.$$

We call S_M valid, *if each procedural Mealy machine* P_M^c *is valid. A procedural Mealy machine* P_M^c *is valid iff it satisfies the following properties:*

- *instrumentation-consistency, i.e,*

$$\forall (q, \widehat{a}, \widehat{o}) \in \lambda^c : \widehat{a} \in (\widehat{\Sigma}_{call} \cup \{\widehat{r}\}) \Rightarrow \widehat{o} \in \{\widehat{👍}, \widehat{👎}\}.$$

- *error-closure, i.e,*

$$\forall q \in Q^c, \widehat{a}_1, \widehat{a}_2 \in \widehat{\Sigma} : \lambda^c(q, \widehat{a}_1) = \widehat{👎} \Rightarrow \lambda^c(\delta^c(q, \widehat{a}_1), \widehat{a}_2) = \widehat{👎}.$$

- *return-closure, i.e,*

$$\forall q \in Q^c, \widehat{a} \in \widehat{\Sigma} : \lambda^c(\delta^c(q, \widehat{r}), \widehat{a}) = \widehat{👎}.$$

- *call-closure, i.e.,*

$$\forall q \in Q^c, \widehat{a} \in \widehat{\Sigma}, \widehat{p} \notin Term : \lambda^c(\delta^c(q, \widehat{p}), \widehat{a}) = \widehat{👎}.$$

Intuitively, *instrumentation-consistency* enforces that all procedural actions, i.e., calling procedures or returning from procedures, must output any of the distinct "success" symbol or "error" symbol. This constraint is necessary for embedding the transduction semantics into behavioral automata via a synchronous alphabet construction (see below). Especially for the return symbol, allowing arbitrary output symbols would result in multiple eligible return symbols which the SBA formalism does not support. One may justify this constraint by the fact that instrumentation itself already requires enhancing a system with "external" symbols that previously had no outputs at all. Therefore, enforcing fixed outputs for these symbols does not impact the transduction semantics of the original system.

Error-closure is the equivalent of the prefix-closure of behavioral automata from a complementary point of view. Rather than requiring all prefixes to be accepted as well, now all continuations of once "rejected" steps (denoted via the "error" output symbol) must continue to exhibit this behavior. *Return-closure* and *call-closure* are direct adaptions from behavioral automata, adjusted to the respective "error" output symbol.

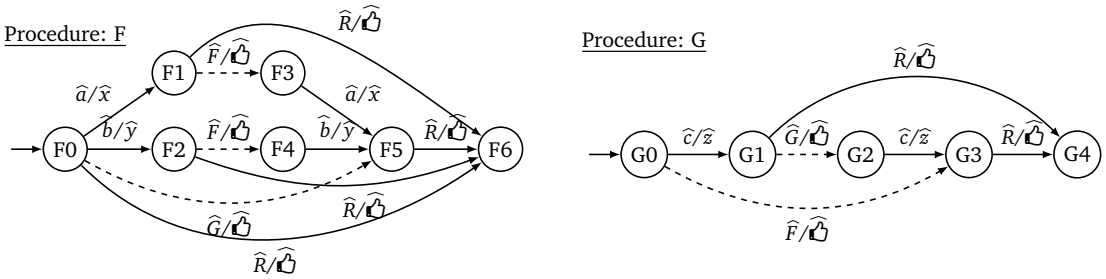
To give an intuition for the structure and properties of SPMMs, consider Figure 3.7 which shows the procedural Mealy machines based on the palindrome system of Figure 3.5.

For modeling the transduction of an SPMM via an SBA, it is necessary to unify the input alphabet and output alphabet of the SPMM into a single alphabet in order to be compatible with the SBA. For this task, we use a synchronous pairing of inputs and outputs via the notion of a *synchronous SPA input alphabet*.

Definition 53 (Synchronous SPA input alphabet)
Let Σ *be an SPA input alphabet and* Ω *be an SPA output alphabet. The* synchronous SPA input alphabet *is a disjoint union* $\Sigma^\times = \Sigma^\times_{call} \uplus \Sigma^\times_{int} \uplus \{r^\times\}$ *over* $\Sigma \times \Omega$ *such that*

- $\Sigma^\times_{call} = \Sigma_{call} \times \{👍\}$,

Figure 3.7
Procedural Mealy machines based on the palindrome system of Figure 3.5 where input a outputs x, input b outputs y, and input c outputs z. Sink states and corresponding transitions (with output 👎) are omitted for readability.



- $\Sigma_{int}^{\times} = \Sigma_{int} \times \Omega_{int}$,
- $r^{\times} = \langle r, 👍 \rangle$.

The semantics-preserving translation of procedural Mealy machines to behavioral automata follows the classic Mealy-to-DFA translation steps with special focus on correctly associating the "error" output symbol with the rejection of DFAs.

Definition 54 (Behavioral chracterization of a procedural Mealy machine)
Let Σ be an SPA input alphabet, Ω be an SPA output alphabet, and $S_M = \langle P_M^c, \ldots \rangle$ be an SPMM over Σ and Ω. The behavioral characterization *of $P_M^c = \langle Q^c, q_0^c, \widehat{\Sigma}, \widehat{\Omega}, \delta^c, \lambda^c \rangle$ for $c \in \widehat{\Sigma}_{call}$ is given by the DFA $P_B^c = \langle Q'^c, q'^c_0, \widehat{\Sigma}^{\times}, Q'^c_F, \delta'^c \rangle$ such that*

- $Q'^c = Q^c$,
- $q'^c_0 = q_0^c$,
- $Q'^c_F = Q^c$,
- $\delta'^c = \{\langle p, \langle \widehat{a}, \widehat{o} \rangle, q \rangle \in Q'^c \times \widehat{\Sigma}^{\times} \times Q'^c \mid \delta^c(p, \widehat{a}) = q, \lambda^c(p, \widehat{a}) = \widehat{o}, \widehat{o} \neq \widehat{👎}\}$.

Note that the above construction does not construct a total or minimal automaton, as it contains undefined transitions and potentially unreachable states. The behavioral automaton can be easily made total and minimal via a post-processing step as discussed in Definition 9.

It is easy to see, how the resulting behavioral automata and consequently the resulting SBA represent all error-free transductions of the original SPMM. For every non-"error" output-labeled transition $\cdot \xrightarrow{a/o} \cdot$, there exists a corresponding $\langle a, o \rangle$-labeled transition in the respective behavioral automaton reaching an accepting state. Every transition with an output 👎 is left undefined so that after totalization and minimization, every one of these transitions lead into a rejecting sink state, representing in SBA semantics that no further call action, internal action, or return action is possible. The requirements on valid procedural Mealy machines (cf. Definition 52) directly ensure that the corresponding behavioral automata are valid as well. The proposed translation directly leads to the following correspondence between the transductions of SPMMs and words of the translated SBAs.

Definition 55 (Transduction of an SPMM)
Let Σ be an SPA input alphabet, Ω be an SPA output alphabet, $S_M = \langle P_M^c, \ldots \rangle$ be an SPMM over Σ and Ω, and $S_B = \langle P_B^c, \ldots \rangle$ be the transformed SBA over $\Sigma^\times$. We define the transductions $T(S_M)$ of S_M as

$$T(S_M) = \{\langle a_1 \cdot \ldots \cdot a_n, o_1 \cdot \ldots \cdot o_n \rangle \in (\Sigma^* \times \Omega^*) \mid \langle a_1, o_1 \rangle \cdot \ldots \cdot \langle a_n, o_n \rangle \in L(S_B)\}.$$

As previously mentioned, the above definition only covers error-free transductions and may not represent a left-total relation. This can be easily compensated for by extending $T(S_M)$ with a global (error-) closure which to any non-mapped input sequence w_a assigns the output sequence of the maximum prefix of w_a that is found in $T(S_M)$ and fills the remaining output symbols (to reach equal length) with repetitions of 👎.

With the semantics of SPMMs defined via SBAs, the question arises whether a native Mealy-based formalism is actual necessary for describing deterministic (instrumented) context-free transductions with a lock-step-based pattern. The increases in comprehensibility and efficiency very much justify this step. Comparing the SPMM based on Figure 3.7 with its respective SBA-based representation, the synchronous alphabet alone results in a lot of "noise" in form of additional input symbols and transitions which mask the essential characteristics of the system. Chapter 10 shows that a native SPMM-based representation of a transduction is much more efficient which, especially in the context of MBT and AAL, allows for a nice boost in performance of the respective techniques.

Note that alternatively, one could try to embed the transduction semantics using alternating input and output symbols that are discussed in Section 2.1.3 as well. However, especially for instrumented systems, the special role of individual symbols would introduce quite a few corner cases. For example, in order to denote that a return symbol can be executed successfully ($\cdot \xrightarrow{\widehat{r}/\widehat{\mathbb{O}}} \cdot$), one would need to return on the output symbol (breaking the semantics of SBAs) or encode the output after the $\widehat{r}$-successor in the calling procedure (making models harder to understand). An experimental implementation of this approach is used in Chapter 10 for comparison and shows no significant benefit.

Furthermore, Section 11.2.1 discusses some possible generalizations of SPMMs that would allow for true extensions that the current SBA-based characterization does not support such as individual return outputs. Here, the extended semantics would further speak for a native transduction formalism.

3.5 Monitors

With SPAs and SBAs, the previous sections present formalisms for (prefix-closed) languages of instrumented context-free systems for which the latter of the two is also used as the foundation of instrumented context-free transductions in the form of SPMMs. The corresponding language definitions (cf. Definitions 29 and 45) characterize the respective languages from a generator-based point of view by describing how words of these languages can be constructed from a given SPA or SBA. As discussed in Section 2.1.2, formal languages may equivalently be characterized from an acceptor-based point of view via a

formalism that parses a word and decides whether it belongs to the language in question.

In this section, we look at acceptor-based approaches for the instrumented languages discussed in this thesis. Specifically, we look at *monitors*, i.e., programs that are given an SPA, SBA, or SPMM and monitor input sequences, e.g., by observing a running system, to decide whether the observed system behavior matches the expected behavior of the SPA, SBA, or SPMM. Monitoring is a powerful tool in practice as it allows one to utilize use-cases from (potentially in-production) systems to supplement the testing of the system. Section 9.3 discusses the impact of monitoring on the MBQA process in more detail.

3.5.1 Monitor-SOS

In order to characterize the general approach to monitoring the concerned systems, we return to SOS because it allows one to focus on functional properties while abstracting from technical implementation details. The central concept is a generic *monitor pattern* that defines a monitor on the basis of a language-SOS.

Definition 56 (Monitor-SOS pattern)
Let Σ be an SPA input alphabet and $\psi \in \{SPA, SBA\}$ denote a model type over Σ. Let $w \in \Sigma^*$ denote a (non-empty) word such that $w = u \cdot v$ for $u \in \Sigma, v \in \Sigma^*$. We define the monitor-SOS *of a monitor Mon as*

$$\frac{(s_i, \sigma_i)_\psi \xrightarrow{u} (s_{i+1}, \sigma_{i+1})_\psi}{(u \cdot v, \bot)_{Mon} \rightarrow (v, \bot)_{Mon}}$$

with $s_i \in \widehat{\Sigma}^*, \sigma_i \in ST(\Gamma_{SPA})$ for all $i \in \{1, \ldots, |w| + 1\}$.

A monitor Mon accepts *a word w, iff*

$$(w, \bot)_{Mon} \rightarrow^* (\varepsilon, \bot)_{Mon}$$

with $s_1 = \widehat{c}_0, \sigma_1 = \bot$.

Intuitively, an input symbol can be successfully parsed iff the corresponding language-SOS emits this symbol beginning from a valid initial configuration. Note that the input sequence in the state of the monitor configuration is processed on a symbol-wise basis. Since the language-SOSs of SPAs and SBAs also emit symbols on a symbol-wise basis, the monitor-SOS also supports an "online" processing style which does not require the full input sequence to be known beforehand. The decision whether the configuration $(\varepsilon, \bot)_{Mon}$ is reached, depends on when the monitor is questioned for a verdict.

While Definition 56 provides a very intuitive characterization of the functionality of a monitor, it provides no guidance on possible implementations. A major challenge for this task is the fact that the respective language-SOSs emit words non-deterministically. Both for SPAs and SBAs, the call-rules of the respective language-SOSs (cf. Definitions 29 and 45) only require the next state to be a member of the respective procedural languages but do not further specify which word to choose. To tackle this problem, one can use the notion of rigorous (de-) composition of SPAs and SBAs (cf. Theorems 1 and 4) to

decompose global words into multiple local words of involved procedures. Since each procedural language is a regular language, one can use the procedural (or behavioral) automata for parsing these languages. This directly allows for an implementation of a global monitor on the basis of a hierarchy of individual regular monitors.

Exploiting the notion of rigorous (de-) composition also highlights the nuances between SPAs and SBAs again. As discussed in Section 3.3, SPAs describe a system holistically, i.e., the procedural traces answer the question whether it is possible to return from a procedural invocation. As a result, the procedural membership question can often[3] only be answered *after* monitoring the complete procedural trace, i.e., after parsing the matching return symbol of an invoked procedure. As a result, the explicit monitor-SOS for SPAs presented in [59] always has to check whether it is still possible to accept the current procedural trace.

In contrast, SBAs allow for a much more fine-grained monitor. Since behavioral automata are prefix-closed, any continuations of rejected words are also rejected. As a result, an SBA-based monitor can truthfully answer the procedural membership question after *every* monitored symbol and therefore check the validity of a call-rule after every parsed input symbol as well. In general, this makes SBAs a much more suited for describing applications that should be monitored.

Due to the SBA-based characterization of SPMMs, a similar result holds for SPMM-based monitors. A native SPMM-based monitor can simply compare the expected output symbol of the SPMM with the monitored output symbol of the system to directly detect mismatches.

3.6 Summary

This section concludes the chapter by summarizing its main results.

- SPAs (Definition 27) are an automaton-based formalism for describing procedural systems. The languages of SPAs (Definition 29) describe arbitrary instrumented (Definition 24) CFLs (Theorem 3).
- SPAs exhibit a notion of rigorous (de-) composition of instrumented global words into procedural local words, which allows for a global-to-local projection (Theorem 1) and a local-to-global expansion (Theorem 2).
- SBAs (Definition 43) describe prefix-closed SPA languages (Definition 45) and support additional features such as non-terminating procedures.
- Similar to SPAs, SBAs also support a notion of rigorous (de-) composition supporting projection (Theorem 4) and expansion (Theorem 5) between instrumented global words and procedural local words.
- SPMMs (Definition 51) are a specialized formalism for instrumented context-free transductions that follow a deterministic lock-step-based pattern. Their semantics are defined via SBAs over a synchronous alphabet of input symbols and output symbol (Definition 55).

[3] One can only ignore the remaining symbols if the DFA enters a (rejecting or accepting) sink state.

- All formalisms may be used in a monitor-based context (Definition 56) for a parser-based interpretation of the languages (transductions).

CHAPTER 4

Model Verification of Instrumented Context-Free Systems

This chapter presents an approach for verifying (the language and transductions of) systems of procedural automata (SPAs), systems of behavioral automata (SBAs), and systems of procedural Mealy machines (SPMMs). This process is based on a translation of the respective models into context-free process systems (CFPSs) [37] and using an existing model checker for CFPSs [162]. Furthermore, Section 7.2 presents the translation of SPAs into visibly push-down automata (VPAs) for which Section 8.1.1 discusses further verification techniques.

4.1 General Notes

Before discussing the suggested translations, one should note that SPAs, SBAs, and SPMMs already provide an innate support for model verification. Due to the notion of rigorous (de-) composition of the three model types into their respective procedural components (cf. Theorems 1 and 4), one can easily use existing model verification techniques for (regular) models to verify inner-procedural behavior. As a result, the following sections specifically focus on the verification of the *global* systems.

4.2 SPAs

The topic of this section is the translation of SPAs into CFPSs [37]. SPAs and CFPSs share a lot of similarities in the sense that both formalisms describe context-free systems that are composed of individual procedural components. CFPSs consist of (multiple) procedural abstractions, called procedural process graphs (PPGs), which can mutually call each other following the classic expansion semantics known from context-free grammars (CFGs). Due to the previously discussed relations between the instrumentation, expansion, and language of SPAs (cf. Section 3.2.5), translating the semantics of SPAs into CFPSs reduces to incorporating the proposed instrumentation into the respective PPGs of the target CFPSs. For this task, the concepts of the construction of instrumented procedural automata (cf. Definition 40) are re-used.

Definition 57 (Induced PPG)
Let Σ be an SPA input alphabet and $P^c = \langle Q^c, q_0^c, \widehat{\Sigma}_{proc}, Q_F^c, \delta^c \rangle$ denote a procedural automaton for $c \in \Sigma_{call}$. We define the induced PPG *for P^c as a tuple $iPPG^c = \langle \Sigma_c, Trans, \rightarrow_c, \sigma_c^s, \sigma_c^e \rangle$ where*

- $\Sigma_c = \{\{start_c\}, \{end_c\}\} \cup \{\{q^c\} \mid q^c \in Q^c\}$ *is a set of state classes,*
- $Trans = Act \cup \mathcal{N}$ *is a set of transformations, where*
  - $Act = \Sigma$ *is a set of actions and*
  - $\mathcal{N} = \Sigma_{call}$ *is a set of names,*
- $\rightarrow_c = \rightarrow_c^{Act} \cup \rightarrow_c^{\mathcal{N}}$ *is the transition relation where*
  - $\rightarrow_c^{Act} = \rightarrow_{start}^{Act} \cup \rightarrow_{int}^{Act} \cup \rightarrow_{end}^{Act}$ *with*
    - $\rightarrow_{start}^{Act} = \{\langle start_c, c, q_0^c \rangle\}$,
    - $\rightarrow_{int}^{Act} = \{\langle q_1, a, q_2 \rangle \mid q_1, q_2 \in Q^c, \widehat{a} \in \widehat{\Sigma}_{int}, \delta^c(q_1, \widehat{a}) = q_2\}$,
    - $\rightarrow_{end}^{Act} = \{\langle q, r, end_c \rangle \mid q \in Q_F^c\}$, *and*
  - $\rightarrow_c^{\mathcal{N}} = \{\langle q_1, a, q_2 \rangle \mid q_1, q_2 \in Q^c, \widehat{a} \in \widehat{\Sigma}_{call}, \delta^c(q_1, \widehat{a}) = q_2\}$,
- $\sigma_c^s = \{start_c\}$,
- $\sigma_c^e = \{end_c\}$.

Intuitively, an induced PPG is a graph-based representation of an instrumented procedural automaton. Most of the structural information is copied, where transitions for internal symbols are translated to *action*-based edges and transitions for call symbols are translated to *name*-based edges which represent expansion points for procedural calls. Furthermore, two additional states, $start_c$ and end_c, are used as designated start node and end node in order to incorporate the instrumentation semantics:

- $start_c$ ensures that every run throughout the graph initially traverses an action-based edge labeled with the corresponding call symbol prior to reaching the node representing the initial state of the procedural automaton, and
- connecting each node that corresponds to an accepting state of the procedural automaton with the end_c node ensures that the PPG has to traverse an r-labeled, action-based edge prior to reaching the end state.

It is easy to see, how this construction directly reflects the notion of *instrumentation* of Definitions 24 and 40. Note that the above construction also ensures that induced PPGs are *terminating* and *guarded* as required by [37].

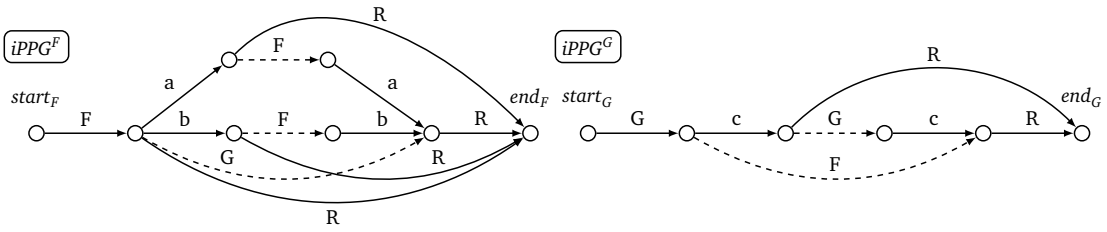
By aggregating the induced PPGs of each procedure of an SPA, one can construct the corresponding induced CFPS.

Definition 58 (Induced CFPS)
Let Σ be an SPA input alphabet and S be an SPA over Σ. We define the induced CFPS *for S as a tuple $iCFPS^S = \langle \mathcal{N}, Act, \Delta, P_0 \rangle$ such that*

- $\mathcal{N} = \Sigma_{call}$ *is a set of names,*
- $Act = \Sigma$ *is a set of actions,*
- $\Delta = \{iPPG^c \mid c \in \Sigma_{call}\}$ *is a finite set of induced PPGs with names in $\mathcal{N}$ and*
- $P_0 = iPPG^{c_0}$ *is the "main" PPG.*

Let $\rightarrow^{\mathcal{N}} = \bigcup_{c \in \Sigma_{call}} \rightarrow_c^{\mathcal{N}}$. We denote by $Exp(iCFPS^S)$ the (potentially infinite state) expansion

Figure 4.1
The induced CFPS of the SPA based on Figure 3.1. Nodes and transitions created from the sink states of the procedural automata are omitted for readability. Action-based edges are depicted via solid lines, name-based edges via dashed lines.



of P_0 in which recursively every name-based edge $\langle p, c, q\rangle \in \rightarrow^{\mathcal{N}}$ is replaced by a copy of $iPPG^c$ such that $p = start_c$ and $q = end_c$ [37].

An example of an induced CFPS is shown in Figure 4.1 which depicts the induced CFPS of the palindrome SPA based on Figure 3.1.

Please note that in [37] the semantics of CFPSs are usually interpreted in the context of an environment and a valuation function $\mathcal{V}\colon AP \to 2^{\Sigma_c}$ which associates with atomic propositions a set of state classes in which they hold. As discussed in Section 2.2, this thesis mainly focuses on (labeled) path-equivalence and therefore implicitly assumes a constant valuation function that returns the empty set. However, in certain situations, users may enhance the induced CFPS with atomic propositions for some convenient shortcuts when specifying requirements.

In the following, we look at the equivalence between words of an SPA language and the existence of a correspondingly labeled path from $start_{c_0}$ to end_{c_0} in the expansion of P_0 of the induced CFPS. This directly enables one to use CFPS model checkers with support for input modalities for the verification of SPA languages.

Theorem 6
Let Σ be an SPA input alphabet, S be an SPA over Σ and $iCFPS^S$ be the induced CFPS of S. Let $w = a_1 \cdot \ldots \cdot a_n \in L(S)$ denote a word of S. Then we have

$$w \in L(S) \Leftrightarrow start_{c_0} \xrightarrow{a_1} \ldots \xrightarrow{a_n} end_{c_0}$$

with $start_{c_0}, \ldots, end_{c_0} \in Exp(iCFPS^S)$.

Proof. The statement directly follows from the construction of induced CFPSs and induced PPGs. By Definition 57, induced PPGs exhibit the same structural properties as instrumented procedural automata (cf. Definition 40). The start edge and the end edge correspond to the $\widehat{c}$-labeled transition and the $\widehat{r}$-labeled transition respectively, the internal edges correspond to the internal transitions, and the named-edges correspond to the $\overline{c}$-labeled transitions (for $\overline{c} \in \overline{\Sigma}_{call}$). This means, for a procedural automaton P^c, its instrumented version $\overline{P}^c$ accepts a word iff there exists a correspondingly labeled path

in $iPPG^c$ from $start_c$ to end_c in which each occurrence of a $\bar{c}$ (for all $\bar{c} \in \bar{\Sigma}_{call}$) traverses a correspondingly labeled name-based edge. By Definition 58, induced CFPSs exhibit the same structural properties as instrumented SPAs (cf. Definition 41). Since the expansion of CFPSs follows the same semantics as Definition 38 and we have identical expansion points (named edges and $\bar{c}$-transitions), Theorem 3 directly concludes the statement where $start_{c_0}$ corresponds to $(\widehat{\bar{c}}_0, \bot)_{Exp}$ and end_{c_0} corresponds to $(\varepsilon, \bot)_{Exp}$. □

Theorem 6 allows one to relate words of an SPA language $L(S)$ to paths in the induced CFPS $iCFPS^S$, that terminate in end_{c_0}. One has to pay special attention to this property, as specifying requirements in temporal logics such as the computational tree logic (CTL) with actions (cf. Definition 17) may often only cover prefixes of paths. In order to address "whole-word" verification, one has to *encode* "termination" of an SPA word via the CTL formula "**AX***false*". This formula is only satisfied by nodes which have no outgoing edges. When constructing induced CFPSs from SPAs with total procedural automata, this formula only holds in end_{c_0} of the initial expansion of P_0 as all other end nodes of nested expansions of induced PPGs coincide with some inner nodes. Consequently, this formula identifies the CFPS state that corresponds to the final structural operational semantics (SOS) configuration $(\varepsilon, \bot)_{Exp}$ and allows one to determine the well-matchedness of the word that is represented by a path.

4.2.1 Examples

To give an intuition for the proposed process of SPA language verification, this (sub-) section presents some exemplary formulae and their satisfiability. Burkart et al. [37] present model checking algorithms for CFPSs and CFPS-like structures [38], that allow one to verify modal μ-calculus formulae for these systems. In the following, we focus on alternation-free μ-calculus formulae which directly cover the set of CTL-expressible requirements including actions (cf. Definition 17). In [162], Steffen and Murtovi present an implementation of the model checker of [37] that is publicly available as a part of AutomataLib (which is a part of LearnLib [95]) and is used for evaluating the formulae of this example.

Table 4.1 shows a set of CTL formulae with actions that are evaluated on the SPA based on Figure 3.1. For this, the SPA is taken and transformed into the corresponding induced CFPS of Figure 4.1 such that the model checker of [162] can be run. For reference, the table also provides the translated modal μ-calculus formulae.

The first formula is violated because there exists no word in the language of the SPA that contains two subsequent bs. Note that while the original palindrome language (cf. Example 2) certainly allows for subsequent bs, e.g., in the palindrome $bbcbb$, the production rules of F can only emit a single b before delegating to another procedure (non-terminal). Since the language-SOS incorporates the instrumentation, this delegation is explicitly observed. By adjusting for this fact, one can see that the corresponding second formula is satisfied.

As previously discussed, "**AX***false*" may be used to encode well-matched SPA words. With the third formula, one checks whether the system allows for finite palindromes. The

Table 4.1
A set of CTL formulae evaluated on the exemplary SPA based on Figure 3.1. In addition, the table shows the μ-calculus formulae as translated by the tool of [162]. Here, the un-parameterized box modalities ($[\,]$) and diamond modalities ($\langle\rangle$) represent the universally quantified and existentially quantified **X** operators, respectively.

	Formula	Description	$\overset{?}{\models}$
1.	$\mathbf{EF}(\langle b\rangle\langle b\rangle true)$ $\mu Z_0.((\langle b\rangle(\langle b\rangle true)) \vee (\langle\rangle Z_0))$	There exists a (sub-)path labeled bb.	✗
2.	$\mathbf{EF}(\langle b\rangle\langle F\rangle\langle b\rangle true)$ $\mu Z_0.((\langle b\rangle(\langle F\rangle(\langle b\rangle true))) \vee (\langle\rangle Z_0))$	There exists a (sub-)path labeled bFb.	✓
3.	$\mathbf{EF}(\mathbf{AX} false)$ $\mu Z_0.(([\,] false) \vee (\langle\rangle Z_0))$	There exists a path to the final state.	✓
4.	$\mathbf{AF}(\mathbf{AX} false)$ $\mu Z_0.(([\,] false) \vee ((\langle\rangle true) \wedge ([\,] Z_0)))$	All paths eventually reach the final state.	✗
5.	$\mathbf{AG}(\mathbf{EF}(\mathbf{AX} false))$ $\nu Z_1.((\mu Z_0.(([\,] false) \vee (\langle\rangle Z_0))) \wedge ([\,] Z_1)))$	On all paths there exists a path to the final state.	✓
6.	$\mathbf{AG}([F](\mathbf{AF}(\langle R\rangle true)))$ $\nu Z_1.(([F](\mu Z_0.((\langle R\rangle true) \vee ((\langle\rangle true) \wedge ([\,] Z_0))))) \wedge ([\,] Z_1)))$	Globally, all F-successors must have an R-successor eventually.	✓

fourth formula is violated because the palindrome system allows for infinite recursion in which, e.g., procedure F, repeatedly invokes itself. Therefore, there exists an infinite path that may not reach the final state. However, with the fifth formula one can verify that at least termination is always possible in the system. This makes perfect sense because one can at any time decide to no longer perform recursive calls.

4.3 SBAs

For verifying instrumented context-free *behaviors* in the form of SBAs, several of the previously presented concepts are re-usable. Recall that the main differences between procedural automata and behavioral automata concern the prefix-closure and the addition of the return symbol to the input alphabet of behavioral automata. Due to the prefix-closure, requirements no longer need to encode well-matched words via the utility formula "**AX***false*". However, the transformation now needs to actively filter out paths of rejected words, which in case of holistic SPAs are implicitly filtered out by not reaching a return edge. Furthermore, due to the inclusion of the return symbol, adding a return edge now needs to be decided on the basis of the acceptance of $\widehat{r}$-successors and not on the basis of the acceptance of the states themselves. These adjustments are formalized in Definition 59.

Definition 59 (Induced behavioral PPG)
Let Σ be an SPA input alphabet and $P_B^c = \langle Q^c, q_0^c, \widehat{\Sigma}, Q_F^c, \delta^c \rangle$ denote a behavioral automaton for $c \in \Sigma_{call}$. We define the induced behavioral PPG *for P_B^c as a tupel $iPPG_B^c = \langle \Sigma_c, Trans, \rightarrow_c, \sigma_c^s, \sigma_c^e \rangle$ such that*

- $\Sigma_c = \{\{start_c\}, \{end_c\}\} \cup \{\{q^c\} \mid q^c \in Q_F^c\}$ *is a set of state classes,*
- $Trans = Act \cup \mathcal{N}$ *is a set of transformations, where*
  - $Act = \Sigma$ *is a set of actions and*
  - $\mathcal{N} = \Sigma_{call}$ *is a set of names,*
- $\rightarrow_c = \rightarrow_c^{Act} \cup \rightarrow_c^{\mathcal{N}}$ *is the transition relation where*
  - $\rightarrow_c^{Act} = \rightarrow_{start}^{Act} \cup \rightarrow_{int}^{Act} \cup \rightarrow_{end}^{Act}$ *with*
    - $\rightarrow_{start}^{Act} = \{\langle start_c, c, q_0^c \rangle\}$,
    - $\rightarrow_{int}^{Act} = \{\langle q_1, a, q_2 \rangle \mid q_1, q_2 \in Q_F^c, \widehat{a} \in \widehat{\Sigma}_{int}, \delta^c(q_1, \widehat{a}) = q_2\}$,
    - $\rightarrow_{end}^{Act} = \{\langle q, r, end_c \rangle \mid q \in Q_F^c, \delta(q, \widehat{r}) \in Q_F^c\}$, *and*
  - $\rightarrow_c^{\mathcal{N}} = \{\langle q_1, a, q_2 \rangle \mid q_1, q_2 \in Q_F^c, \widehat{a} \in \widehat{\Sigma}_{call}, \delta^c(q_1, \widehat{a}) = q_2\}$,
- $\sigma_c^s = \{start_c\}$,
- $\sigma_c^e = \{end_c\}$.

The main differences between Definition 57 and Definition 59 concern the set of state classes and the notion of $\rightarrow_{end}^{Act}$. For the construction of Σ_c (and the transitions between the state classes) only the set of accepting states Q_F^c is considered to prevent including paths which correspond to rejected words. Furthermore, the construction of $\rightarrow_{end}^{Act}$ no longer requires the source state to be accepting but the source state to have an accepting $\widehat{r}$-successor. When applying this transformation to an SBA such as the one described

in Figure 3.5, we see that the induced behavioral PPG coincides with the one shown in Figure 4.1. Here, however, the sink nodes and corresponding edges are not omitted for readability but actually do not exist. Otherwise, the switch to behaviors does not significantly impact the structural properties of induced behavioral PPGs compared to (plain) induced PPGs. Especially, the notion of *induced behavioral CFPS* is similar to Definition 58 and therefore omitted here.

As previously mentioned, the utility formula "**AX***false*" is no longer needed to denote the well-matchedness of words (paths). As seen in formulae one, two and six of Table 4.1, one can easily and intuitively describe prefixes of words as well so that the adaption of requirements to SBAs and their respective behaviors is straightforward.

4.4 SPMMs

For the verification (of the transductions) of SPMMs, it is possible to exploit its characterization via SBAs (cf. Section 3.4). By transforming SPMMs into SBAs first, the concepts of Section 4.3 can be directly applied to verify the transduction steps of the original SPMMs. While the SBA-based interpretation introduces some overhead such as the larger input alphabet, this only affects the verification process but not the specification of requirements in, e.g., CTL. A transduction step such as $\cdot \xrightarrow{a/o} \cdot$ in the SPMM context can be directly represented by an $\langle a, o \rangle$ input modality in the SBA context.

4.5 Summary

This section concludes the chapter by summarizing its main results.

- The verification of SPA languages can be implemented via a (path-) equivalent translation into CFPSs, which allows one to use existing model checkers for the actual verification process. This embedding requires a special formula to correctly encode well-matched words.
- The verification of SBA languages only requires a minor adjustment of the construction of the respective CFPSs (to correctly incorporate the semantics of termination and reachability) and stays identical to the SPA case otherwise. Especially for the specification of requirements, it is no longer necessary to actively encode well-matched words.
- The verification of SPMMs is based on the translation of SPMMs into equivalent SBAs as proposed in Section 3.4, which directly enables one to utilize the presented SBA verification for the verification of transductions.

CHAPTER 5

Model-Based Testing of Instrumented Context-Free Systems

This chapter presents the construction of conformance tests for systems of procedural automata (SPAs), systems of behavioral automata (SBAs), and systems of procedural Mealy machines (SPMMs). The topic involves general concepts for the conformance testing of SPAs as well as necessary adjustments for SBA-based and SPMM-based conformance testing using the example of the W-method [44].

5.1 General Concepts

The notion of rigorous (de-) composition is a central concept for the construction of conformance tests of SPA, SBA, and SPMM. According to Theorems 1 and 4, the languages of SPAs and SBAs (and via a translation to SBAs, the transductions of SPMMs as well) are fully characterized by the languages of their respective procedural components. Therefore, by verifying the conformance of each model-procedure with its corresponding implementation-procedure, one can automatically conclude the conformance of the global SPA, SBA, and SPMM model with the respective implementation. Since the procedures of SPAs, SBAs, and SPMMs are represented by regular automata, i.e., (prefix-closed) deterministic finite acceptors (DFAs) or Mealy machines, existing concepts from regular conformance testing can be used to construct these procedural tests. The main topics of the following sections then concern the execution of these local conformance tests on the global system and the handling of potential challenges of this process.

Note that by decomposing the global conformance tests into procedural conformance tests and delegating the constructions thereof to existing approaches, the global conformance tests are subject to the same assumptions and guarantees as the procedural ones. This is especially important for the aspects of correctness and completeness. Since the black-box equivalence problem is impossible to solve even for simplest formalisms such as DFAs [126], it is not possible to provide any better results for SPAs, SBAs, or SPMMs. Instead, the following sections focus on involving the procedural conformance tests in the most generic way possible in order to allow for the transfer of assumptions and guarantees for regular systems to instrumented context-free systems. This should allow one to easily modify the construction of global conformance tests (by using difference approaches for constructing the procedural conformance tests) in order to meet one's individual needs.

5.2 SPAs

In order to evaluate procedural conformance tests on a global system, Theorem 2 states that the membership property of a single local word of a procedure (of an SPA) can be evaluated on the global SPA system by expanding the local word via the gamma expansion (cf. Definition 36) and embedding the expanded word in a valid context. However, this process requires access sequences, terminating sequences, and return sequences of the involved procedures. Therefore, the construction of an SPA conformance test is a two-step process. First, the computation of access sequences, terminating sequences, and return sequences for the involved procedures of a given SPA model and the validation thereof on the implementation. Second, the construction of the global conformance test via the union of the individual expanded procedural conformance tests. The following two (sub-) sections present an algorithmic approach for implementing these two steps.

5.2.1 Computing Access Sequences, Terminating Sequences, and Return Sequences

Algorithm 5.1 sketches the algorithmic approach for extracting access sequences, terminating sequences, and return sequences for a given SPA model. In essence, the algorithm operates in two separate phases. The first phase extracts a terminating sequence for each procedure, whereas the second phase extracts a pair of matching access sequence and return sequence for each procedure. Both phases follow the structure of a fix-point computation.

First Phase

The first phase, from Line 2 to Line 17, starts with initializing some global variables. The set of currently eligible alphabet symbols Σ_{cur} is set to the internal alphabet of the given SPA input alphabet Σ, the set of finished procedures Σ_{fin} for which a terminating sequence is already computed is set to the empty set, and the boolean flag *stable* for indicating a fix-point is set to **false**.

The main computation loop of the first phase, from Line 7 to Line 16, iterates over the currently unfinished procedures and uses Dijkstra's single-source-shortest-path algorithm [49] to compute paths from the initial state to the accepting states of the currently investigated procedure c, using only the set of currently eligible symbols $\widehat{\Sigma}_{cur}$. Here, the paths are assumed to be encoded via the words that transition P^c from its initial state q_0^c to an accepting state q_f^c, therefore $SP \subseteq L(P^c)$. If there exists at least one path (cf. Line 9), an arbitrary one is selected and its expansion is stored as the terminating sequence. Note that the computation of shortest paths is restricted to $\widehat{\Sigma}_{cur}$ so that all terminating sequences required for the expansion process are available to γ. Since all procedures are reachable in minimal SPAs, there exists at least one terminating sequence for every procedure and by the finiteness of words of SPA languages there exists at least one terminating sequence that only consists of internal symbols, representing the start of the fix-point computation. Furthermore, by Theorem 2, $\gamma(\widehat{w})$ constitutes a valid terminating sequence for c.

Algorithm 5.1
Computation of access sequences, terminating sequences, and return sequences of SPAs.

Input: A minimal SPA S over a given SPA input alphabet Σ
Output: The values of as_c, ts_c, and rs_c for each $c \in \Sigma_{call}$

function COMPUTEASTSRS(S, Σ)
  $\Sigma_{cur} \leftarrow \Sigma_{int}$
  $\Sigma_{fin} \leftarrow \emptyset$
  stable $\leftarrow$ **false**
  while not *stable* **do**
    stable $\leftarrow$ **true**
    for $c \in (\Sigma_{call} \setminus \Sigma_{fin})$ **do**
      $SP \leftarrow \bigcup_{q_f^c \in Q_F^c}$ DIJKSTRASSSP$(q_0^c, \widehat{\Sigma}_{cur}, q_f^c)$
      if $SP \neq \emptyset$ **then**
        $\widehat{w} \leftarrow$ CHOOSE(SP)
        $ts_c \leftarrow \gamma(\widehat{w})$
        $\Sigma_{cur} \leftarrow \Sigma_{cur} \cup \{c\}$
        $\Sigma_{fin} \leftarrow \Sigma_{fin} \cup \{c\}$
        stable $\leftarrow$ **false**
      end if
    end for
  end while
  $as_{c_0} \leftarrow c_0$
  $rs_{c_0} \leftarrow r$
  $\Sigma_{fin} \leftarrow \{c_0\}$
  stable $\leftarrow$ **false**
  while not *stable* **do**
    stable $\leftarrow$ **true**
    for $c \in \Sigma_{fin}$ **do**
      for $\widehat{w} \in \left(\left(\bigcup_{k=0}^{|Q^c|} \widehat{\Sigma}_{proc}^k\right) \cap L(P^c)\right)$ **do**
        for $i \in \{1, \ldots, |\widehat{w}|\} : w[i] \in (\Sigma_{call} \setminus \Sigma_{fin})$ **do**
          $as_{w[i]} \leftarrow as_c \cdot \gamma(\widehat{w}[, i-1]) \cdot w[i]$
          $rs_{w[i]} \leftarrow r \cdot \gamma(\widehat{w}[i+1,]) \cdot rs_c$
          $\Sigma_{fin} \leftarrow \Sigma_{fin} \cup \{w[i]\}$
          stable $\leftarrow$ **false**
        end for
      end for
    end for
  end while
  return $\{\langle as_c, ts_c, rs_c \rangle \mid c \in \Sigma_{call}\}$
end function

Afterwards, the set of eligible alphabet symbols and finished procedures is extended by c. In subsequent iterations of the processing loop (cf. Line 7) this means that the algorithm no longer investigates procedures for which a terminating sequence has already been computed and that a call symbol c becomes eligible for computing subsequent terminating sequences only after a terminating sequence for procedure c itself has been found, ensuring correct expansions.

The management of the fix-point indicator *stable* ensures that whenever a new terminating sequence is discovered, the set of procedures for which no terminating sequence has been computed yet is re-investigated. Note that we only consider minimal SPAs (cf. Definition 30) which ensures that each procedure accepts a non-empty language and each call symbol is reachable from the main procedure of S. Consequently, the step-wise extension of Σ_{cur} ensures that the algorithm always finds a shortest path to an accepting state eventually and therefore computes a valid terminating sequence for all procedures of the given SPA model.

Second Phase

The second phase, from Line 18 to Line 34, starts with storing the trivial pair of access sequence and terminating sequence of the main procedure c_0 and re-initializing the set of finished procedures accordingly. Similar to the first phase, the boolean flag *stable* indicates an (initially unstable) fix-point throughout the main computation loop.

The loop from Line 24 to Line 33 iterates over the set of finished procedures, i.e., procedures which provide an admissible context because access sequences and return sequences have already been found for them. Each of these procedures are analyzed regarding accepted paths up to length $|Q^c|$, again, encoded as words over the respective alphabet. Line 26 scans each path for occurrences of call symbols that have not yet been added to the set of finished procedures. For each such call symbol $w[i]$, the concatenation of the access sequence of the currently investigated procedure c, the expanded prefix $\widehat{w}[,i-1]$, and the call symbol $w[i]$ itself constitutes a valid access sequence for $w[i]$ and is stored as such. Since the currently investigated path $\widehat{w}$ reaches an accepting state, the concatenation of the return symbol, the expanded suffix of $\widehat{w}[i+1,]$, and the return sequence of the currently inspected procedure c also constitutes a valid return sequence for the current procedure. Afterwards, $w[i]$ is added to the set of finished procedures and a re-evaluation is triggered by setting the fix-point flag to **false**.

Note that the outer loop (cf. Line 24) only iterates over finished procedures. Therefore, the referenced access sequences as_c and return sequences rs_c are always well-defined. Since the new access sequence and return sequence are (in part) constructed from the same accepted local word, the new pair of sequences forms a valid context as well.

Compared to the first phase, the second phase traverses the call-hierarchy in reverse order, starting from the initial procedure and traversing the hierarchy to the inner-most calls. Again, the minimality of S ensures that each procedure is involved in at least one accepted word of S so that this step-wise extension process successfully computes a pair of (matching) access sequence and return sequence for each procedure of S.

5.2.2 SPA Conformance Test

In order to use the extracted sequences in the construction of an SPA conformance test, it is necessary to verify whether they constitute actual access sequences, terminating sequences, and return sequences of the implementation as well. By Definition 35, this can be simply verified by testing whether the concatenations of the three sequences are accepted by the implementation. Consequently, these concatenations can be seen as the initial tests of the SPA conformance test. We continue with the introduction of a utility notation to later reference these tests.

Definition 60 (Set of extracted access sequences, terminating sequences, return sequences)
Let Σ be an SPA input alphabet and S be an SPA over Σ. We define the set of extracted access sequences, terminating sequences, and return sequences *as follows:*

$$EATR(S) = \{as_c \cdot ts_c \cdot rs_c \mid \langle as_c, ts_c, rs_c \rangle \in \textsc{computeEASTSRS}(S, \Sigma)\}.$$

Implementing the idea of Section 5.1, the SPA conformance test can be directly constructed from the union of the (expanded) individual conformance tests. The correctness of this approach (with respect to the assumptions about the conformance tests for the procedural automata) directly follows from Theorem 1.

Definition 61 (SPA conformance test)
Let Σ be an SPA input alphabet and S be an SPA over Σ. We define the SPA conformance test *as*

$$CT(S) = EATR(S) \cup \left\{ \bigcup_{c \in \Sigma_{call}} \{as_c \cdot \gamma(\widehat{w}) \cdot rs_c \mid \widehat{w} \in CT(P^c)\} \right\}$$

where $CT(P^c)$ denotes the conformance test for procedure P^c of S and the respective access sequences, terminating sequences, and return sequences are taken from the results of Algorithm 5.1.

Theorem 7 (SPA conformance)
Let Σ be an SPA input alphabet, S^{mod} be a valid SPA model over Σ, and S^{impl} be an (unknown) valid SPA implementation over Σ. $CT(S^{mod})$ is a conformance test for S^{mod}, i.e.,

$$(\forall w \in CT(S^{mod}) \colon w \in L(S^{mod}) \Leftrightarrow w \in L(S^{impl})) \Rightarrow S^{mod} \equiv_{SPA} S^{impl}$$

with respect to the assumptions about the procedural conformance tests.

Proof. This is a direct consequence of the notion of rigorous (de-) composition of SPAs. By Theorem 1, we know that the global languages of S^{mod} and S^{impl} are characterized by the local languages of the involved procedures. By Definition 19, a conformance test can test the equivalence of (in this case) DFAs with respect to the assumptions about regular conformance tests. By Theorem 2, these tests of the procedural conformance tests can be answered by the (global) SPA implementation with the chosen expansion in Definition 61. By aggregating the individual conformance tests and verifying the

extracted access sequences, terminating sequences, and return sequences we can verify the (language-) equivalence of each individual procedural automaton which directly concludes the (language-) equivalence of the concerned SPAs (cf. Corollary 1). □

5.3 SBAs

In analogy to SPAs, Theorem 5 provides a means to evaluate the local behavior of behavioral automata on the global system that they constitute. Therefore, the construction of conformance tests for SBAs follows a similar approach to Section 5.2: By computing the respective sequences required by the expansion process and using expanded conformance tests of the (regular) behavioral automata, the global conformance between a model and an implementation can be tested. However, both, the computation of sequences and the computation of the procedural conformance tests need to be adjusted to the specific semantics of SBAs. The following (sub-) sections discuss these adjustments.

5.3.1 Computing Access Sequences and Terminating Sequences

Regarding sequence computation, one major difference between SPAs and SBAs is the fact that procedural automata (of SPAs) encode termination with the acceptance of states, whereas behavioral automata (of SBAs) use explicit return-transitions into accepting states. As a result, the computation of terminating sequences needs to be adjusted. Rather than computing paths to arbitrary accepting states, only states whose $\widehat{r}$-successors are accepting must be considered. By Definition 44, behavioral automata of valid SBAs only accept words containing at most one return symbol and only if it constitutes the last symbol of the word. Consequently, when expanding these words in the call-rules of the language-structural operational semantics (SOS) of SBAs, they emit well-matched subsequences (well-matched with respect to the expanded call symbol) and therefore their expansions represent valid terminating sequences for the respective procedure.

A second major difference between SPAs and SBAs is the fact that SBAs describe prefix-closed languages. On the one hand, this allows for non-terminating procedures, i.e., procedures for which no terminating sequences exist. On the other hand, by Theorem 5, evaluating procedural tests on the global system only requires access sequences and terminating sequences of the involved procedures which allows one to skip the computation of return sequences.

Algorithm 5.2 summarizes the above changes in an algorithmic notation. Since it operates in a similar fashion to Algorithm 5.1, we focus only on the changes needed to address the SBA semantics. Line 4 explicitly denotes the set of terminating procedures in order to restrict the computation of terminating sequences to them (cf. Line 8). Furthermore, the reachability analysis is restricted to states with accepting $\widehat{r}$-successors (cf. Lines 9 and 10). The computation of access sequences (from Line 20 onward) remains identical to Algorithm 5.1 with only the computation of return sequences removed. The result of the algorithm (cf. Line 35) is updated to account for non-terminating procedures which do not posses terminating sequences.

Algorithm 5.2
Computation of access sequences and terminating sequences of SBAs.

Input: A minimal SBA S_B over a given SPA input alphabet Σ
Output: The values of as_c and ts_c (if available) for each $c \in \Sigma_{call}$

function COMPUTEASTS(S_B, Σ)
  $\Sigma_{cur} \leftarrow \Sigma_{int}$
  $\Sigma_{fin} \leftarrow \emptyset$
  $\Sigma_{term} \leftarrow \{c \in \Sigma_{call} | \exists \widehat{w} \in L(P_B^c) \colon \widehat{w}[|\widehat{w}|] = \widehat{r}\}$
  stable $\leftarrow$ **false**
  while not *stable* **do**
    stable $\leftarrow$ **true**
    for $c \in (\Sigma_{term} \setminus \Sigma_{fin})$ **do**
      $Q^c_{term} = \{q \in Q^c \mid \delta^c(q, \widehat{r}) \in Q^c_F\}$
      $SP \leftarrow \bigcup_{q \in Q^c_{term}}$ DIJKSTRASSSP$(q^c_0, \widehat{\Sigma}_{cur}, q)$
      if $SP \neq \emptyset$ **then**
        $\widehat{w} \leftarrow$ CHOOSE(SP)
        $ts_c \leftarrow \gamma(\widehat{w})$
        $\Sigma_{cur} \leftarrow \Sigma_{cur} \cup \{c\}$
        $\Sigma_{fin} \leftarrow \Sigma_{fin} \cup \{c\}$
        stable $\leftarrow$ **false**
      end if
    end for
  end while
  $as_{c_0} \leftarrow c_0$
  $\Sigma_{fin} \leftarrow \{c_0\}$
  stable $\leftarrow$ **false**
  while not *stable* **do**
    stable $\leftarrow$ **true**
    for $c \in \Sigma_{fin}$ **do**
      for $\widehat{w} \in \left(\left(\bigcup_{k=0}^{|Q^c|} \widehat{\Sigma}^k_{proc}\right) \cap L(P_B^c)\right)$ **do**
        for $i \in \{1, \ldots, |\widehat{w}|\} \colon w[i] \in (\Sigma_{call} \setminus \Sigma_{fin})$ **do**
          $as_{w[i]} \leftarrow as_c \cdot \gamma(\widehat{w}[, i-1]) \cdot w[i]$
          $\Sigma_{fin} \leftarrow \Sigma_{fin} \cup \{w[i]\}$
          stable $\leftarrow$ **false**
        end for
      end for
    end for
  end while
  return $\{\langle as_c, ts_c \rangle \mid c \in \Sigma_{term}\} \cup \{\langle as_c \rangle \mid c \in (\Sigma_{call} \setminus \Sigma_{term})\}$
end function

5.3.2 SBA Conformance Test

Before discussing the details of SBA conformance tests, we look at the notion of extracted access sequences and terminating sequences.

Definition 62 (Set of extracted access sequences and terminating sequences)
Let Σ be an SPA input alphabet and S_B be an SBA over Σ. We define the set of extracted access sequences and terminating sequences *as follows:*

$$EAT(S_B) = \{as_c \cdot ts_c \cdot r \mid \langle as_c, ts_c \rangle \in \textsc{computeASTS}(S_B, \Sigma)\} \cup \{as_c \mid \langle as_c \rangle \in \textsc{computeASTS}(S_B, \Sigma)\}$$

Note that due to skipping the return sequences and computing the terminating sequences based on $\widehat{r}$-successors, it is necessary to append an additional return symbol in order to verify the correctness of the extracted terminating sequences. For validating the access to non-terminating procedures, it suffices to test the respective access sequences. This construction aligns with Definition 47.

Recall that for SPAs, one can simply use the (local) conformance tests of the involved procedures and use the union of the expanded words as a conformance test for the (global) SPA system. However, SBAs introduce a major challenge for this concept by including the return symbol in the input alphabet of behavioral automata. Transferring the approach of SPAs to SBAs leads to problems with the decidability of conformance tests of behavioral automata.

To give an intuition for this problem, consider how, e.g., the W-method [44] (cf. Section 2.3) constructs a conformance test: Each word of a transition cover set is concatenated with each word of a characterizing set. Since behavioral automata contain $\widehat{r}$-transitions, the transition cover set necessarily contains words that traverse these $\widehat{r}$-transitions. Concatenating these words with any non-empty element from the characterizing set results in (local) test words that extend beyond an initial occurrence of $\widehat{r}$. Not only is the expansion of such words (specifically the return symbol) undefined (cf. Definition 36), but also any continuation beyond a (local) return symbol escapes the scope of a procedure in the global context. In general, this leads to non-predictable behavior because the response of the global system depends on the access sequence of the procedure of which the local conformance test has no knowledge of. Similarly, calls to non-terminating procedures also cannot be expanded correctly because they do not have terminating sequences.

For tackling these issues, this thesis proposes a set of adjustments for the construction of the procedural conformance tests. In the following, these adjustments are sketched via an adapted version of the W-method [44] which constructs a conformance test from the cartesian product of a transition cover set and a (state) characterizing set. Other approaches for computing conformance tests need to be adjusted individually, but the ideas remain the same.

For the problem of traversing “critical” transitions in the transition cover set, i.e., successors of non-terminating call-transitions or successors of return-transitions, the computation of the transition cover set is modified. First, the procedural input alphabet

$\widehat{\Sigma}$ is partitioned into "non-continuable" input symbols $\widehat{\Sigma}_{ncon}$, i.e., call symbols to non-terminating procedures and the return symbol as well as "continuable" input symbols $\widehat{\Sigma}_{cont}$, i.e., the rest. Then, the computation of the state cover set is restricted to $\widehat{\Sigma}_{cont}$ and two distinct transition cover sets are computed depending on the input symbols that are used to extend the state cover set.

Definition 63 ((Non-) continuable transition cover sets)
Let Σ be an SPA input alphabet that is partitioned into "non-continuable" input symbols $\widehat{\Sigma}_{ncon}$ and "continuable" input symbols $\widehat{\Sigma}_{cont}$ with respect to an SBA S_B over Σ. Let P_B^c denote a behavioral automaton of S_B for $c \in \Sigma_{call}$. We define the "non-continuable" transition cover set *nTCS and the* "continuable" transition cover set *cTCS of a behavioral automaton P_B^c as*

$$nTCS(P_B^c) = SCS_{\widehat{\Sigma}_{cont}}(P_B^c) \cdot \widehat{\Sigma}_{ncon},$$
$$cTCS(P_B^c) = SCS_{\widehat{\Sigma}_{cont}}(P_B^c) \cdot \widehat{\Sigma}_{cont}.$$

By the construction of the respective transition cover sets, one can directly conclude the following properties.

Lemma 5
Let Σ be an SPA input alphabet and S_B be an SBA over Σ. Let P_B^c denote a behavioral automaton of S_B for $c \in \Sigma_{call}$.

1. *$nTCS(P_B^c)$ only contains words that end with call symbols to non-terminating procedure or the return symbol.*
2. *$cTCS(P_B^c)$ does not contain any words with call symbols to non-terminating procedure or the return symbol.*

Proof. This is a direct consequence of Definition 63. □

The union of the two transition cover sets almost resembles the classic transition cover set over the full alphabet $\widehat{\Sigma}$ (cf. Definition 22). The only exceptions are outgoing transitions of states that are only reachable via non-continuable symbols. However, by the call-closure and return-closure of valid behavioral automata (cf. Definition 44), all these uncovered transitions must lead into a single rejecting sink state. In particular, the union of the two transition cover sets cover all remaining transitions and reaches all states.

The construction of the characterizing set does not require any modifications. Due to the return-closure and call-closure of valid behavioral automata, any continuations beyond non-continuable input symbols transfer the behavioral automata into a sink state. As a result, any word that extends beyond the initial occurrence of a non-continuable symbol cannot distinguish any states as the observable behavior is always identical (rejection). Hence, these words cannot be part of any characterizing set of behavioral automata. The following lemma summarizes this property.

Lemma 6
Let Σ be an SPA input alphabet and S_B be an SBA over Σ. For each behavioral automaton P_B^c of S_B, the characterizing set of P_B^c, $CS(P_B^c)$, contains no words that extend beyond a call symbol of a non-terminating procedure or a return symbol.

Proof. This is a direct consequence of the return-closure and call-closure of P_B^c (cf. Definition 44). Any continuation of a word that traverses a return-transition or a call-transition of a non-terminating procedure transitions the behavioral automaton into a rejecting sink state, making it impossible to distinguish behavior beyond this input. □

What remains to be discussed is how the transition cover sets and the characterizing set need to be connected. In analogy to the original W-method [44], one constructs the cartesian product of the continuable transition cover set and the characterizing set but excludes the non-continuable transition cover set from this process. Note that the purpose of appending elements from the characterizing set is to distinguish the state that is reached by the respective word of the transition cover set from all other states. For non-continuable transitions, this is no longer necessary in valid behavioral automata. Here, the characterization of the state is achieved by external constraints (call-closure and return-closure of valid behavioral automata) and mutual characterization across the involved procedures of the SBA, i.e., verifying that a procedure is indeed non-terminating. For all remaining (continuable) transitions, the proposed construction resembles the original W-method [44], inheriting its properties (and requiring its assumptions) regarding correctness and completeness.

Definition 64 (SBA conformance test)
Let Σ be an SPA input alphabet and S_B be an SBA over Σ with behavioral automata P_B^c for $c \in \Sigma_{call}$. Let $\gamma' : (\widehat{\Sigma}^*_{proc} \cdot \widehat{\Sigma}_{ncon}) \to \Sigma^*$ denote the adjusted expansion function *that expands words with the exception of the last symbol, i.e.,*

$$\gamma'(\widehat{w}) = \gamma(\widehat{w}[, |\widehat{w}| - 1]) \cdot w[|\widehat{w}|].$$

We define the (W-method-based) SBA conformance test *as*

$$CT(S_B) = EAT(S_B) \cup \left\{ \bigcup_{c \in \Sigma_{call}} \{as_c \cdot \gamma'(\widehat{w}) \mid \widehat{w} \in nTCS(P_B^c) \cup ((cTCS(P_B^c) \cup \{\varepsilon\}) \cdot CS(P_B^c))\} \right\}.$$

Note that the adjusted expansion function is just a shorthand notation for the default SBA expansion of Theorem 5. Furthermore, by Lemmas 5 and 6 only the last symbol passed to γ' may be a call symbol of a non-terminating procedure or a return symbol. Therefore, all calls to γ are well-defined. The inclusion of ε in the construction of the cartesian product with the characterizing set is due to the special semantics of the W-method (cf. Section 2.3). One can then show, that a conformance test constructed according to Definition 64 constitutes a proper conformance test for a given SBA.

Theorem 8 (SBA conformance)
Let Σ be an SPA input alphabet, S_B^{mod} be a valid SBA model over Σ, and S_B^{impl} be an (unknown) valid SBA implementation over Σ. $CT(S_B^{mod})$ is a conformance test for S_B^{mod}, i.e.,

$$(\forall w \in CT(S_B^{mod}) : w \in L(S_B^{mod}) \Leftrightarrow w \in L(S_B^{impl})) \Rightarrow S_B^{mod} \equiv_{SBA} S_B^{impl}$$

with respect to the (W-method-based) assumptions about the procedural conformance tests.

Proof. This is a direct consequence of the notion of rigorous (de-) composition of SBAs. By Theorem 4, we know that the global languages of S_B^{mod} and S_B^{impl} are characterized by the local languages of the involved behavioral automata. For transitions labeled with continuable input symbols, the construction of procedural conformance tests coincides with the W-method. For transitions with non-continuable inputs, the behavior is defined by external constraints (cf. Definition 44). As a result, the procedural conformance tests can ensure conformance with respect to the assumptions about the original algorithm (here, the W-method [44]). By Theorem 5, these tests of the procedural conformance tests can be answered by the (global) SBA implementation with the adjusted expansion used in Definition 64. By aggregating the individual conformance tests and verifying the extracted access sequences and (if available) terminating sequences one can verify the (language-) equivalence of each behavioral automaton which directly concludes the (language-) equivalence of the concerned SBAs (cf. Corollary 2). □

5.3.3 Example

To give an intuition for the construction of an SBA conformance tests, let us look at the computation described in Definitions 62 and 64 for the SBA based on Figure 3.5. For the computation of the extracted access sequences and terminating sequences, it is easy to see that both behavioral automata P_B^F and P_B^G describe terminating procedures. Therefore, a potential set of extracted access sequences and terminating sequences is given by

$$EAT(S_B) = \{\langle F, \varepsilon\rangle, \langle F \cdot a \cdot G, \varepsilon\rangle\},$$

where $\langle F, \varepsilon\rangle$ denotes the pair of extracted sequences for P_B^F and $\langle F \cdot a \cdot G, \varepsilon\rangle$ denotes the pair of extracted sequences for P_B^G.

For the computation of the local conformance tests, we focus on the steps for P_B^F as the process for P_B^G is analogous. First, the input alphabet is partitioned into "non-continuable" and "continuable" input symbols. We have

$$\widehat{\Sigma}_{ncon} = \{\widehat{R}\} \text{ and}$$
$$\widehat{\Sigma}_{cont} = \{\widehat{a}, \widehat{b}, \widehat{c}, \widehat{F}, \widehat{G}\}.$$

A possible state cover set on the basis of $\widehat{\Sigma}_{cont}$ is given by

$$SCS_{\widehat{\Sigma}_{cont}}(P_B^F) = \{\varepsilon, \widehat{a}, \widehat{b}, \widehat{c}, \widehat{G}, \widehat{a} \cdot \widehat{F}, \widehat{b} \cdot \widehat{F}\}$$

which covers all states except F6, including the omitted sink state (via $\widehat{c}$). The respective transition cover sets are constructed by

$$nTCS(P_B^F) = SCS_{\widehat{\Sigma}_{cont}}(P_B^F) \cdot \widehat{\Sigma}_{ncon}$$
$$cTCS(P_B^F) = SCS_{\widehat{\Sigma}_{cont}}(P_B^F) \cdot \widehat{\Sigma}_{cont}$$

which cover all transitions except the outgoing transitions of F6. A possible characterizing set for P_B^F is given by

$$CS(P_B^F) = \{\varepsilon, \widehat{a}, \widehat{b}, \widehat{F}, \widehat{F} \cdot \widehat{a}, \widehat{R}\}.$$

The local conformance test for P_B^F is then given by

$$CT(P_B^F) = nTCS(P_B^F) \cup ((cTCS(P_B^F) \cup \{\varepsilon\}) \cdot CS(P_B^F)).$$

With the exception of test words that extend beyond non-continuable input symbols, this local conformance test covers the same characteristics that the regular W-method would cover. Note that the properties of successors of transitions labeled with non-continuable input symbols, e.g., F6, are still checked by $nTCS(P_B^F)$. The only transitions (and potentially the sink state) that are omitted by the proposed conformance tests are the ones that have their behavior defined by the external constraints of validity (cf. Definition 44), e.g., the outgoing transitions of F6. Therefore, these transitions (and the sink state) do not need to be tested if the implementation is a valid SBA as well. Furthermore, by looking at the structure of the individual test words, one sees how the adjusted expansion function γ' (cf. Definition 64) is able to expand the words without any problems.

5.4 SPMMs

The construction of conformance tests for SPMMs is based on the equivalent characterization of the concerned transductions as SBAs languages. Using the suggested techniques to represent deterministic transductions that follow an incremental lock-step pattern via a prefix-closed language over the cartesian product of input symbols and output symbols, the concepts and results of Section 5.3 directly apply to SPMMs. Therefore, from a qualitative point of view, there is no further investigation into the construction of conformance tests for SPMMs needed.

However, it is worth noting that many of the algorithms for constructing (regular) conformance test (cf. Section 8.2) are originally designed for Mealy machines or can be easily adapted to natively work on Mealy machines. In practice, this allows one to skip any transformation or mapping layer as discussed in Section 2.1.3 and directly work with *native* Mealy-based models. As a consequence, the resulting characterizing sets may be smaller because they can exploit the explicit semantics of Mealy machines. However, the overall approach to constructing a conformance test for SPMMs remains identical to the SBA case.

5.5 Summary

This section concludes the chapter by summarizing its main results.

- The notion of rigorous (de-) composition of the (global) behavior of SPAs, SBAs, and SPMMs into the (local) behaviors of their respective procedures allows one to construct conformance tests for the global systems from the aggregation of conformance tests for the local procedures.
- For SPAs, this involves

1. the computation of access sequences, terminating sequences, and return sequences on the basis of the SPA model and
2. the computation of procedural (regular) conformance tests for the individual procedures of the model.

After verifying the extracted sequences, they are eligible for the expansion process of Theorem 2 in order to evaluate the union of expanded (local) conformance tests on the (global) SPA implementation.

- For SBAs, the process is similar but requires slight adjustments to correctly handle the distinct notion of termination of behavioral automata and the fact that SBAs may contain non-terminating procedures. For the computation of (local) conformance tests of behavioral automata, one can
  1. omit the computation of return sequences because they are not necessary for the expansion process and
  2. omit the verification of successors of non-continuable transitions because their behavior is defined externally via constraints of valid behavioral automata.

  After verifying the extracted sequences, they are eligible for the expansion process of Theorem 5 in order to evaluate the adjusted local conformance tests on the (global) SBA implementation.
- The construction of conformance tests for SPMMs follows directly from the previous discussions about representing lock-step-based transductions via prefix-closed languages (cf. Section 3.4). Here, computing *native* Mealy-based conformance tests may boost the performance of the conformance testing process by reducing the length of the tests.

CHAPTER 6

Active Automata Learning of Instrumented Context-Free Systems

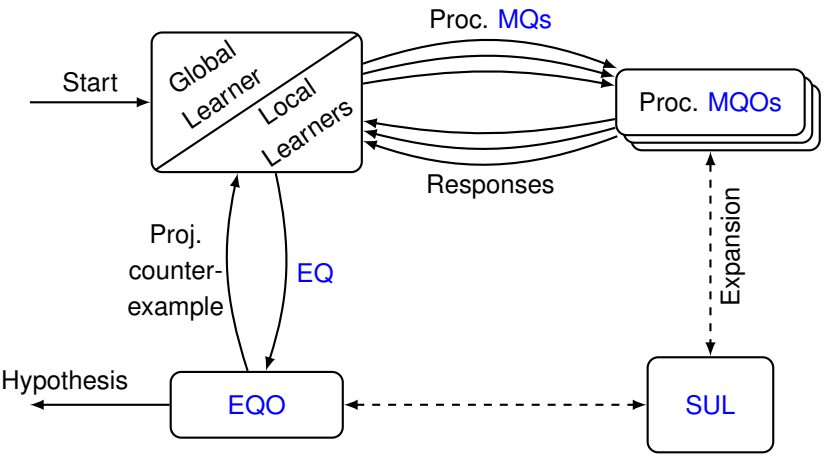
This chapter presents an algorithmic approach for inferring models of systems of procedural automata (SPAs), systems of behavioral automata (SBAs), and systems of procedural Mealy machines (SPMMs) in the context of the minimally adequate teacher (MAT) framework. This approach implements the inference of the global models on the basis of a simultaneous inference of their involved procedural components. Furthermore, this chapter analyzes the correctness and complexity of the inference processes and presents several heuristics for improving their practical performance.

6.1 General Concepts

Key to learning SPA-based, SBA-based, and SPMM-based systems is the notion of rigorous (de-) composition presented in Sections 3.2.4 and 3.3.2. As Theorems 1 and 4 state, the (global) behavior of SPAs, SBAs, and (via embedding in SBAs) SPMMs is alternatively characterized by the behavior of the individual (local) procedures. Therefore, the problem of learning SPA-based, SBA-based, and SPMM-based systems can be re-interpreted as a problem of learning their respective procedures. In all cases, the concerned procedures are represented by deterministic finite acceptors (DFAs) or Mealy machines, for which (regular) active automata learning (AAL) algorithms exist and to which this task can be delegated. Consequently, the main task for SPA, SBA, and SPMM learners can be streamlined to organizing a simultaneous inference of individual procedures and taking care of properly transforming information between the global, instrumented system under learning (SUL) and the individual local learning algorithms.

The concepts of expansion and projection play an essential role in this process as the two techniques directly allow one to map information between the global view and the local views of a system. As discussed in Section 2.4, many MAT-based learning algorithms alternate between an *exploration* phase and a *verification* phase: During the exploration phase a learner poses membership queries (MQs) to explore the properties of a system and during the verification phase a potential counterexample is provided to the learner in order to refine the current system hypothesis. Figure 6.1 shows how the concepts of expansion and projection lift the regular AAL loop of Figure 2.1 to the level of SPA-based, SBA-based, and SPMM-based systems.

Figure 6.1
The regular AAL loop of Figure 2.1 tailored towards the model types of SPAs, SBAs, and SPMMs.



The SPA, SBA, and SPMM learners presented in this chapter act as managers of various local (regular) learners for the involved procedures and delegate the actual inference processes to them. Due to the results of Theorems 2 and 5, the regular learners can be provided with special procedural membership query oracles (MQOs) that answer the local MQs of the regular learners on the instrumented SUL using the presented expansions. Once the local learners have conjectured their procedural hypotheses, the global hypotheses for the respective formalisms are constructed. As a consequence of Theorems 1 and 4, counterexamples to these global hypotheses can be reduced to counterexamples of at least one involved procedure. Therefore, in contrast to the expansion of local MQs, the global counterexamples of the equivalence query oracle (EQO) are projected to local counterexamples of individual procedures. Since the refinement of procedural hypotheses is part of the regular learning loop, it can be delegated to the regular learners of the concerned procedures as well.

By formalizing these concepts with a special emphasis on a general, language-based characterization, it is possible to employ arbitrary (MAT-compatible) regular AAL algorithms for inferring the individual procedures. This means that the global SPA, SBA, and SPMM learner instances can be *parameterized* with its regular learner instances, which lifts the notion of rigorous (de-) composition from the model level to the learning level. This is a particularly interesting aspect, as Chapter 10 shows that the properties of regular AAL algorithms transfer to the context-free level. In practice, this allows for a fine-grained adjustability to individual needs.

In accordance with the structure of the AAL loop, the following sections present in detail the implementations of the exploration phases and verification phases of the learning algorithms for SPAs, SBAs, and SPMMs.

6.2 SPAs

We continue with the exploration phase and verification phase of the SPA learner and look at the algorithmic properties of the presented approach. Throughout this section, H_{SPA} denotes the current SPA hypothesis that is constructed from the individual procedural hypotheses of the respective local learners. Furthermore, note that in the context of AAL, formal languages are often seen from a parser-based point of view (answering queries, etc.). Therefore, SPAs are referred to in a parser-based interpretation as well, e.g., when talking about SPAs *accepting* a word (cf. Section 3.5).

6.2.1 Exploration Phase

As sketched in Section 6.1, the exploration phase of the SPA learner essentially consists of the simultaneous exploration phases of regular learners for the involved procedures. Given Theorem 2, each MQ of a procedural learner can be answered by the instrumented SUL after expansion. However, for this expansion to work, one requires access sequences, terminating sequences, and return sequences for the involved procedures. At the start of the inference process, these sequences are unknown to the SPA learner and therefore local MQs cannot be expanded properly.

Note that this problem is two-fold: The lack of (matching) access sequences and return sequences means that it is not possible to embed a local MQ in a global context, i.e., a procedural learner cannot pose any MQs at all. The lack of terminating sequences means that a local MQ cannot contain any procedural calls to the respective procedure because they cannot be expanded properly. The proposed approach tackles these problems via the concepts of *deferred learner activation* and *incremental alphabet extension*.

For both concepts, positive counterexamples play an important role. Positive counterexamples are words $ce \in WM(\Sigma)$ which are accepted by the SUL but rejected by the current SPA hypothesis H_{SPA}. Therefore, any call symbol $c \in \Sigma_{call}$ that occurs in a positive counterexample allows one to extract a valid access sequence, terminating sequence, and return sequence of c.

For tackling the problem of missing contexts (access sequences and return sequences), the activation of procedural learners is *deferred*. When constructing the tentative SPA hypothesis H_{SPA}, the leaner constructs a hypothesis that rejects any words that contain call symbols of non-active learners. This forces counterexamples that contain call symbols of non-active learners to be positive, providing access sequences and return sequences of the involved procedures. Upon extracting the respective sequences from the counterexample, the SPA learner *activates* the respective local learners of the concerned procedures and uses their actual procedural hypotheses for subsequent constructions of H_{SPA}. This means for the start of the learning loop, the initial hypothesis of the SPA learner is an empty hypothesis that rejects all words $w \in WM(\Sigma)$ so that the first counterexample guarantees an access sequence, terminating sequence, and return sequence for at least the initial procedure c_0.

This process is synchronized with the proposed incremental alphabet extension. During the learning process, the learner maintains a set of *active alphabet symbols*, denoted

$\Sigma_{act} \subseteq \Sigma_{proc}$, that keeps track of the alphabet symbols that local learners are allowed to use to explore the properties of the procedures. This set is initialized with Σ_{int} and extended by a call symbol c only after a terminating sequence ts_c is observed via a positive counterexample because only then the gamma expansion is able to properly expand local MQs containing cs. Therefore, the procedural hypotheses are partial automata initially (with respect to the procedural SPA input alphabet Σ_{proc}) and become total throughout the learning process as terminating sequences are discovered. Since the global SPA hypothesis H_{SPA} rejects any calls to non-active procedures, the (partial) procedural hypotheses are never queried for undefined information.

Note that this decision makes intermediate hypotheses non-minimal (or rather only minimal with respect to Σ_{act}, cf. Definition 30) which needs some special treatment if the intermediate hypotheses are directly used by other processes (cf. Section 9.4). Other than that, the two concepts do not cause any problems with the individual procedural inference processes. All local learners operate independently of each other and (regular) AAL is monotone with respect to alphabet extension.

6.2.2 Verification Phase

During the verification phase, the SPA learner receives a counterexample that indicates a mismatch between the behavior of the SUL and the current SPA hypothesis H_{SPA}. In the context of SPAs, there exist two kinds of counterexamples: *Positive counterexamples* are words $ce \in WM(\Sigma)$ which are accepted by the SUL but wrongfully rejected by H_{SPA}, whereas *negative counterexamples* are words that are rejected by the SUL but wrongfully accepted by H_{SPA}.

The following discussions generalize the two cases by only distinguishing between an *accepting SPA* $S_A = \langle P_A^{c_1}, \ldots, P_A^{c_n} \rangle$ and a *rejecting SPA* $S_R = \langle P_R^{c_1}, \ldots, P_R^{c_n} \rangle$. In case of a positive counterexample, i.e., if $ce \in L(SUL)$ and $ce \notin L(H_{SPA})$, the SUL represents the accepting system and the SPA hypothesis represents the rejecting system, i.e., $S_A = SUL$ and $S_R = H_{SPA}$. In case of a negative counterexample, i.e., if $ce \notin L(SUL)$ and $ce \in L(H_{SPA})$, the SUL represents the rejecting system and the SPA hypothesis represents the accepting system, i.e., $S_R = SUL$ and $S_A = H_{SPA}$. This generalization shows that the counterexample analysis process is symmetrical for both positive counterexamples and negative counterexamples as only the mapping of the SUL and H_{SPA} to S_A and S_R changes but not the analysis process itself.

The following corollary presents the general approach of analyzing SPA counterexamples.

Corollary 3 ([61])
Let Σ be an SPA input alphabet, $ce \in WM(\Sigma)$ be a counterexample and S_A, S_R be two SPAs over Σ such that $ce \in L(S_A)$ and $ce \notin L(S_R)$. Theorem 1 states that

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

and via negation

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

These two properties directly suggest a two-step approach for refining H_{SPA}:

1. In the *global* step, the SPA learner analyzes *ce* to determine a (not necessarily unique) procedure P_R^* of S_R that rejects its respective projected trace and
2. in the *local* step, the corresponding procedural learner uses the projected sub-trace as a local counterexample for the affected procedure.

If *ce* is a positive counterexample, i.e., if $S_A = SUL$ and $S_R = H_{SPA}$, S_A describes the correct behavior. Here, P_A^* accepts the projected trace and P_R^* should behave the same way. Consequently, the projected trace constitutes a *positive* local counterexample for P_R^* of H_{SPA}. If *ce* is a negative counterexample, i.e., if $S_R = SUL$ and $S_A = H_{SPA}$, S_R describes the correct behavior. Here, P_R^* rejects the projected trace and P_A^* should behave the same way. Consequently, the projected trace constitutes a *negative* local counterexample for P_A^* of H_{SPA}.

The following discussions refer to this counterexample extraction as a function called ANALYZECOUNTEREXAMPLE that for a given counterexample returns a tuple containing the concerned procedure P_R^* or P_A^* (identified via their respective call symbol) as well as the projected local counterexample trace $\alpha(ce[i+1, \rho_{ce}(i+1)])$. We continue with a sketch of the global SPA refinement process to highlight its interaction with the exploration phase and an efficient implementation for the ANALYZECOUNTEREXAMPLE function afterwards.

Algorithmic Sketch for Handling Global Counterexamples

Algorithm 6.1 sketches the main refinement process of the SPA learner. Before discussing its details, recall from Section 6.2.1 that positive counterexamples are crucial for extracting access sequences, terminating sequences, and return sequences, which are required for activating the individual procedural learners and expanding calls to the respective procedures. Furthermore, recall that the initial SPA hypothesis represents an empty system that rejects all input words. Consequently, the SPA learner receives a positive counterexample before any local learners are activated.

From Line 2 to Line 16 the learner specifically handles the case of a positive counterexample. By Definition 29, all words in the language of an SPA begin with the initial procedure. Consequently, in Line 3, one can truthfully identify the initial procedure of the SUL from the first symbol of any positive counterexample. As previously discussed, the set Σ_{act} keeps track of the currently eligible alphabet symbols (including activated call symbols) and is initialized with Σ_{int}. Therefore, this set can be used in the check of Line 4 to identify any previously unobserved procedures. If such a procedure is detected, the counterexample directly provides the respective access sequence, terminating sequence, and return sequence. Line 5 to Line 7 extract the sequences and Line 8 removes the procedure from future analysis.

In Line 11, the algorithm updates the procedural learners according to the new information available.

- Procedural learners that are not active due to the lack of access sequences and return sequences are now activated and begin with the inference of the respective procedures according to the usual AAL workflow.

Algorithm 6.1 (from [61])
Refinement step of the SPA learner that extracts access sequences, terminating sequences, and return sequences from positive counterexamples and delegates a projected local counterexample to the respective procedural learner.

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

1: **function** REFINESPAHYPOTHESIS($ce, answer$)
2: **if** *answer* **then**
3: $c_0 \leftarrow ce[1]$
4: **for all** $c \in$ DETECTNEWPROCEDURES(ce, Σ_{act}) **do**
5: $as_c \leftarrow$ EXTRACTACCESSSEQUENCE(ce, c)
6: $ts_c \leftarrow$ EXTRACTTERMINATINGSEQUENCE(ce, c)
7: $rs_c \leftarrow$ EXTRACTRETURNSEQUENCE(ce, c)
8: $\Sigma_{act} \leftarrow \Sigma_{act} \cup \{c\}$
9: **end for**
10: **for all** $c \in (\Sigma_{call} \cap \Sigma_{act})$ **do**
11: UPDATEPROCEDURALLEARNER(c, Σ_{act})
12: **end for**
13: **if** $ce \in L(H_{SPA})$ **then**
14: **return**
15: **end if**
16: **end if**
17: $\langle c, \widehat{ce}_{local} \rangle \leftarrow$ ANALYZECOUNTEREXAMPLE(ce)
18: REFINEPROCEDURALHYPOTHESIS($P^c, \widehat{ce}_{local}, answer$)
19: **end function**

- Procedural learners that already are active synchronize their input alphabet with the current set of Σ_{act}, which may involve posing queries containing new call symbols that are now properly expandable.

After relaying the information about the respective sequences to the procedural learners, every learner for a procedure $c \in (\Sigma_{call} \cap \Sigma_{act})$ is active and all local hypotheses of the concerned procedures are total automata with respect to the current set of Σ_{act}.

Since the detection of new access sequences, terminating sequences, and return sequences affects the procedural hypotheses and therefore the current SPA hypothesis H_{SPA}, Line 13 checks whether the initial counterexample *ce* still is a valid (positive) counterexample for the (potentially updated) global hypothesis. If this is not the case, the refinement step is terminated early. Otherwise, the algorithm continues with the proposed way of handling global counterexamples. Line 17 extracts the local counterexample on the basis of Corollary 3 and Line 18 delegates the actual refinement process to the local learner of the concerned procedure.

It is worth noting that the SPA learner does not depend on any specific properties of the regular learner(s), which re-iterates the fact that one can use arbitrary (MAT-compatible) regular learning algorithms for inferring SPAs. Section 6.2.4 shows that these steps guarantee progress in the (global) hypothesis progression towards inferring the final SPA model. For now, we continue with looking at an efficient implementation for the ANALYZECOUNTEREXAMPLE method.

Efficient Analysis of Global Counterexamples

For efficiently analyzing SPA counterexamples, it is possible to pursue an approach similar to the one of Rivest et al. [146] for regular systems. In their work, the authors replace a prefix of a given counterexample with an access sequence of the current hypothesis' state reached by the prefix. By checking different lengths of prefixes in a binary search fashion and evaluating the responses of the SUL to the constructed queries, the authors can pinpoint an input symbol that transitions the current hypothesis and the SUL in provably different states and therefore trigger a hypothesis refinement.

The first part of the two-step analysis process derived from Corollary 3 only requires one to identify a rejecting procedure. Therefore, for applying the concepts of Rivest and Schapire to the case of SPAs, it is sufficient to perform a similar style of binary search over the procedural calls (or rather the respective returns) of a counterexample. For a given return symbol in a counterexample, the analysis process needs to transform the prefix up to this return symbol such that it only contains procedural invocations that are guaranteed to be admissible in S_R. By comparing the behavior of S_R with the expected behavior of the counterexample, one is able to determine a procedure of S_R that rejects its respective projected trace and continue to extract the local counterexample.

To formally define this process, we look at the notion of *ts-conformance* and the *alpha-gamma transformation*.

Definition 65 (ts-conformance [61])
Let Σ be an SPA input alphabet and S be an SPA over Σ. Let TS denote a set of terminating

sequences for various procedures of S. We call S ts-conform *with respect to TS iff the language-SOS system of S satisfies the following property:* $\forall ts_c \in TS\colon \exists \widehat{w} \in L(P^c), \sigma \in ST(\Gamma_{SPA})\colon$

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

The analysis process requires that S_R is ts-conform with respect to the set of terminating sequences that are used by the gamma expansion. In the context of the SPA learner, this means the terminating sequences that are extracted from positive counterexamples (cf. Line 6, Algorithm 6.1). It is easy to see that in the case of negative counterexamples this property trivially holds, as S_R maps to the SUL and all terminating sequences are extracted from accepted words of the SUL. In case of positive counterexamples, H_{SPA} needs to be ts-conform with respect to the extracted terminating sequences. The following lemma states, how this property can be checked on SPAs.

Lemma 7 ([61])
Let Σ be an SPA input alphabet, S be an SPA over Σ, and TS be a set of terminating sequences for various procedures of S. Let $ets_c = c \cdot ts_c \cdot r$ denote the embedded terminating sequence *for each $ts_c \in TS$.*

$$S \text{ is ts-conform wrt. } TS \iff \forall ts_c \in TS\colon \forall \langle c', i\rangle \in Inst_{ets_c}\colon \alpha(ets_c[i+1, \rho_{ets_c}(i+1)]) \in L(P^{c'}).$$

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

Verifying ts-conformance of the current SPA hypothesis H_{SPA} can be done by simply checking whether the projections of the currently extracted terminating sequences (including nested invocations) are accepted by the involved procedures. If there exists a (nested sub-) sequence that is not accepted by a procedural hypothesis, the projected sequence directly constitutes a positive local counterexample. This positive counterexample is valid because it is constructed from an extracted terminating sequence, i.e., an accepted (sub-) word of the SUL. By refining the affected procedural hypotheses via standard means of AAL, the procedures (and consequently H_{SPA}) can be made ts-conform with respect to the extracted terminating sequences. Recall that Algorithm 6.1 always activates procedural learners before analyzing a positive counterexample via the call to the AnalyzeCounterexample method (which requires the ts-conformance) so that there always exists a correctly initialized procedural learner for processing the potential local counterexamples.

We continue with the introduction of the *alpha-gamma transformation*.

Definition 66 (Alpha-gamma transformation [61])
Let Σ be an SPA input alphabet and $w \in WM(\Sigma)$. We define

$$[\![\cdot]\!]\colon WM(\Sigma) \to WM(\Sigma)$$

as

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

We generalize $[\![\cdot]\!]$ to (minimally) return-matched words $w_{mrm} \in MRM(\Sigma)$ and obtain a transformation

$$[\![\cdot]\!]^* : MRM(\Sigma) \to MRM(\Sigma)$$

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

$$[\![w_{mrm}]\!]^* = [\![c_{i_1} \cdot w_1 \cdot \ldots \cdot c_{i_n} \cdot w_n]\!]^* = c_{i_1} \cdot [\![w_1]\!] \cdot \ldots \cdot c_{i_n} \cdot [\![w_n]\!].$$

Note that if w_{mrm} happens to be (minimally) well-matched, $[\![\cdot]\!]^*$ coincides with $[\![\cdot]\!]$.

For a given well-matched word, the alpha-gamma transformation replaces every procedural invocation with a terminating sequence of the respective procedure. If an SPA is ts-conform with respect to the terminating sequences used by γ, this means that the transformed word only contains nested calls that are accepted by the respective procedures. The generalized alpha-gamma transformation is used to transform prefixes of counterexamples to prefixes that are guaranteed to not contain any nested calls that are rejected by a ts-conform SPA. This transformation introduces a notion of *acceptance monotonicity* that allows one to define a binary-search style analysis in order to identify a rejecting procedure of S_R similar to the approach of Rivest et al. [146]. Theorem 9 formalizes this property.

Note that the following discussion requires the existence of at least two return symbols in a counterexample. If the counterexample only contains a single return symbol, i.e., the counterexample exposes an in-equivalence in the main procedure, no counterexample analysis is required, as the rejecting procedure is trivially given by c_0.

Theorem 9 (Acceptance monotonicity of $[\![\cdot]\!]^*$ [61])
Let Σ be an SPA input alphabet, $w \in WM(\Sigma)$, and S be an SPA over Σ that is ts-conform with respect to the terminating sequences used by γ. Furthermore, let r_h, r_k be indices of return symbols of w with $r_h < r_k$. Then we have

$$[\![w[,r_h]]\!]^* \cdot w[r_h+1,] \in L(S) \Rightarrow [\![w[,r_k]]\!]^* \cdot w[r_k+1,] \in L(S).$$

Proof. This implication is based on the fact that for all admissible words $v \in MRM(\Sigma)$, $[\![v]\!]^*$ is also admissible in a ts-conform SPA. Furthermore, the admissibility of a word is decided on call-rules of the language-structural operational semantics (SOS), since they are the only rules which are guarded by the procedural membership question. 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 $w[r_k+1,]$ do neither. For the full proof, see Theorem 2 in [61]. □

On the basis of the acceptance monotonicity of $[\![\cdot]\!]^*$, it is now possible to define a binary search for identifying the rejecting procedure of S_R. 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). This means that there exists a decomposition in-between for which the acceptance flips. The analysis process investigates various return symbol indices r_i of the counterexample by asking membership questions

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

If the decomposition is accepted by S_R, the process continues the analysis with a lower return index than r_i because by Theorem 9 the answers to all decompositions at higher return indices are already known. If the decomposition is rejected by S_R, the process continues the analysis with a higher return index than r_i because by contraposition of Theorem 9 the answers to all decompositions at lower return indices are already known.

The process terminates when the lowest return index r_l is found such that

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

Let c_l denote the matching call symbol of r_l and i_l denote its index, i.e., $\rho_{ce}(i_l+1)+1 = r_l$ and $ce[i_l] = c_l$. Since r_l is the lowest return index such that the decomposition is accepted by S_R, i.e., a decomposition with an even lower index is rejected, the call symbol $ce[i_l]$ in the un-processed counterexample was not emitted because

$$\alpha(ce[i_l+1, r_l-1]) \notin L(P^{c_l}).$$

This property directly identifies a rejecting procedure of S_R and allows one to extract a local counterexample. Figure 6.2 visualizes this binary search-style counterexample analysis process by contrasting the two scenarios of investigating higher and lower return indices.

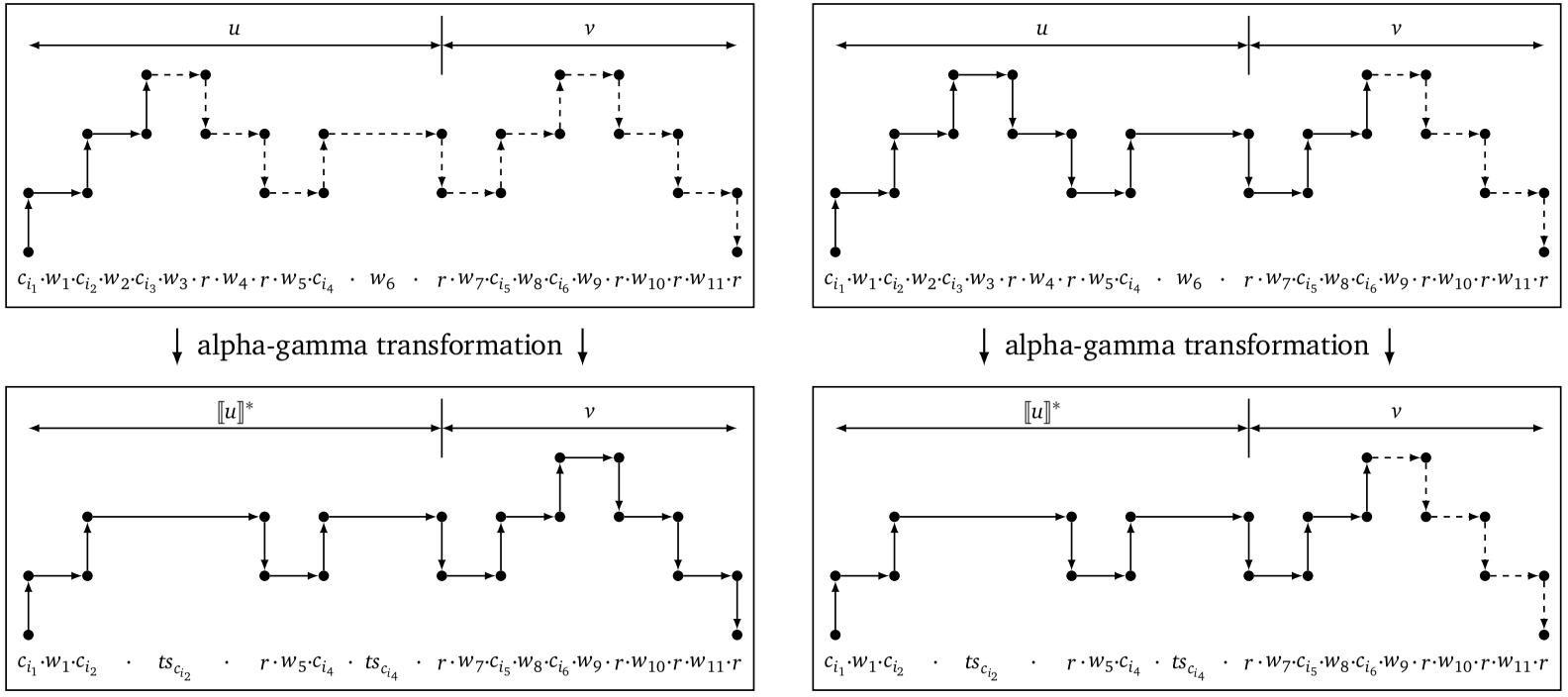
Note that determining i_l and r_l requires one to pose MQs on S_R. In case of negative counterexamples, S_R maps to the SUL and therefore, these MQs result in actual queries to the system. However, in case of positive counterexamples, S_R maps to H_{SPA} and therefore, these MQs can be answered by the (in-memory) hypothesis H_{SPA}. Consequently, positive counterexamples can be analyzed and processed without any (query) costs.

6.2.3 Example

To give an intuition for the learning process, this (sub-) section presents the first iterations of the SPA learner for inferring a SUL based on Figure 3.1.

Start As described in Section 6.2.1, the SPA learner initially has no knowledge of any procedures or access sequences, terminating sequences, or return sequences thereof. As a result, the learner cannot activate any procedural learners and the first hypothesis model H^0_{SPA} resembles the empty SPA which describes the empty language. Other than that, only global variables such as Σ_{act} are initialized accordingly.

Figure 6.2 (from [61])
The two possible scenarios during the analysis of counterexamples: The top two images each show a run of a counterexample 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 site the error occurs in u. Here, the (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 site the error occurs in v. Here, the error prevails even after the (extended) alpha-gamma transformation, which indicates that further analysis, i.e., binary search, should continue with splitting v.



First Counterexample As the first counterexample, the SPA learner may receive the tuple $\langle F \cdot a \cdot R, true \rangle$. This constitutes a positive counterexample because $F \cdot a \cdot R \in L(SUL)$ and $F \cdot a \cdot R \notin L(H^0_{SPA})$. The SPA learner continues with analyzing the counterexample according to Algorithm 6.1. From the positive counterexample, the SPA learner infers that F identifies the initial procedure and stores this information in c_0 accordingly. Given the current valuation of Σ_{act}, the detection of new procedures (cf. Line 4 to Line 9) determines F as a new procedure and stores the respective sequences accordingly. The counterexample yields $as_F = F, ts_F = a, rs_F = R$. With the availability of the new sequences, Line 11 now starts the procedural learner for P^F over $\Sigma_{act} = \{F, a, b, c\}$. The resulting procedural hypothesis (which also constitutes the subsequent SPA hypothesis H^1_{SPA}) is shown in Figure 6.3a. Note that the hypothesis model for P^G is still missing because the respective procedural learner is not yet activated. Furthermore, P^F has no $\widehat{G}$-transitions due to a missing terminating sequence for procedure G.

Activating the procedural learner of P^F and constructing H^1_{SPA} from the hypothesis model of P^F also changes the behavior of the SPA hypothesis compared to the initial empty hypothesis H^0_{SPA}. Most importantly, the initial counterexample $\langle F \cdot a \cdot R, true \rangle$ is no longer a valid counterexample for the current SPA hypothesis because $F \cdot a \cdot R \in L(H^1_{SPA})$. As a result, the refinement step terminates early at the check of Line 13 and the counterexample is handled successfully.

Second Counterexample As the second counterexample, the SPA learner may receive the tuple $\langle F \cdot G \cdot F \cdot b \cdot R \cdot R \cdot R, true \rangle$. This constitutes a positive counterexample because $F \cdot G \cdot F \cdot b \cdot R \cdot R \cdot R \in L(SUL)$ and $F \cdot G \cdot F \cdot b \cdot R \cdot R \cdot R \notin L(H^1_{SPA})$ due to G not being an element of Σ_{act}. Again, the SPA learner continues with analyzing the counterexample according to Algorithm 6.1. Given the current valuation of Σ_{act}, the detection of new procedures (cf. Line 4 to Line 9) determines G as a new procedure and stores the respective sequences accordingly. The counterexample yields $as_G = F \cdot G, ts_G = F \cdot b \cdot R, rs_G = R \cdot R$. Updating the procedural learners on the basis of the new active alphabet symbols (cf. Line 11) results in adding the call symbol G to the procedural learner of P^F (due to the now available ts_G) as well as the activation of the procedural learner of P^G over $\Sigma_{act} = \{F, G, a, b, c\}$ (due to the now available as_G, ts_G, and rs_G). The resulting procedural hypotheses (which constitute the subsequent SPA hypothesis H^2_{SPA}) are shown in Figure 6.3b.

Both procedures are now total with regard to $\widehat{\Sigma}_{proc}$ of Figure 3.1. Similar to the first counterexample, the activation of the procedural learner of P^G (and consequently the construction of its procedural hypothesis model) affects the SPA hypothesis such that the second counterexample is no longer a valid counterexample for H^2_{SPA}. As a result, the refinement step terminates early with the check of Line 13 again.

Third Counterexample As the third counterexample, the SPA learner may receive the tuple $\langle F \cdot a \cdot F \cdot a \cdot R \cdot a \cdot R, true \rangle$. Again, this constitutes a positive counterexample because $F \cdot a \cdot F \cdot a \cdot R \cdot a \cdot R \in L(SUL)$ and $F \cdot a \cdot F \cdot a \cdot R \cdot a \cdot R \notin L(H^2_{SPA})$. Contrary to the first two counterexamples, this counterexample does not contain any new information about

previously unobserved procedures. Therefore, there are no preliminary updates to the current SPA hypothesis and Algorithm 6.1 proceeds with Line 17 in order to analyze the counterexample. Since the given counterexample is a positive one, S_A maps to the SUL and S_R maps to H^2_{SPA}. For the analysis of the counterexample, S_R, i.e., H^2_{SPA}, needs to be ts-conform with respect to the currently stored terminating sequences. For $ts_F = a$, P^F needs to accept the word $\widehat{a}$. For $ts_G = F \cdot b \cdot R$, P^G needs to accept the (projected) word $\widehat{F}$ and due to the nested call within the terminating sequence, P^F needs to accept the word $\widehat{b}$ as well. H^2_{SPA} satisfies these properties as seen in Figure 6.3b.

The analysis process continues with applying the alpha-gamma transformation in a binary search style pattern in order to detect a violating procedure P^*_R. Starting with the return index $r_i = 5$, the transformation yields

$$[\![F \cdot a \cdot F \cdot a \cdot R]\!]^* \cdot a \cdot R = F \cdot [\![a \cdot F \cdot a \cdot R]\!] \cdot a \cdot R = F \cdot a \cdot F \cdot a \cdot R \cdot a \cdot R$$

which represents the original counterexample that is rejected by S_R (H^2_{SPA}). Here, the error still preserves after the transformation and according to Theorem 9, only higher return indices need to be investigated for further information. Analyzing the return index $r_i = 7$ results in the (trivial) replacement of the counterexample with the terminating sequence of procedure F, i.e.,

$$[\![F \cdot a \cdot F \cdot a \cdot R \cdot a \cdot R]\!]^* = [\![F \cdot a \cdot F \cdot a \cdot R \cdot a \cdot R]\!] = F \cdot a \cdot R,$$

which is accepted by the (ts-conform) S_R. As a result, the lowest return index r_l such that the transformed counterexample is accepted by S_R is given by $r_l = 7$ which directly identifies F (the matching call symbol) as the violating procedure and $\widehat{a} \cdot \widehat{F} \cdot \widehat{a}$ as the (projected) local counterexample. Line 18 delegates this local counterexample to the respective procedural learner of P^F for its refinement. The result of this process (and consequently, the updated SPA hypothesis H^3_{SPA}) is shown in Figure 6.3c.

Fourth Counterexample As the fourth counterexample, the SPA learner may receive the tuple $\langle F \cdot G \cdot c \cdot a \cdot F \cdot R \cdot R \cdot R, \mathit{false} \rangle$. This constitutes a negative counterexample because $F \cdot G \cdot c \cdot a \cdot F \cdot R \cdot R \cdot R \notin L(\mathit{SUL})$ and $F \cdot G \cdot c \cdot a \cdot F \cdot R \cdot R \cdot R \in L(H^3_{SPA})$. As Algorithm 6.1 skips the search for new sequences for negative counterexamples, the refinement step directly proceeds with Line 17 in order to analyze the counterexample. Since the given counterexample is a negative one, S_R maps to the SUL and S_A maps to H^3_{SPA}. Since the terminating sequences used by γ are extracted from accepted runs of the SUL, S_R is trivially ts-conform with respect to these sequences.

Regarding the analysis of the counterexample using the alpha-gamma transformation, the algorithm may start with the return index $r_i = 7$. The resulting transformation yields

$$[\![F \cdot G \cdot c \cdot a \cdot F \cdot R \cdot R]\!]^* \cdot R = F \cdot [\![G \cdot c \cdot a \cdot F \cdot R \cdot R]\!] \cdot R = F \cdot G \cdot F \cdot b \cdot R \cdot R \cdot R.$$

Here, the transformation "swallows" the error occurring in P^G by replacing the original invocation with its terminating sequence. This effect is detectable due to the ts-conformance of S_R, which makes it accept the transformed counterexample. By the results of Theorem 9,

the binary search continues to analyze the (only) lower return index $r_i = 6$. The resulting transformation yields

$$[\![F \cdot G \cdot c \cdot a \cdot F \cdot R]\!]^* \cdot R \cdot R = F \cdot G \cdot [\![c \cdot a \cdot F \cdot R]\!] \cdot R \cdot R = F \cdot G \cdot c \cdot a \cdot F \cdot a \cdot R \cdot R \cdot R$$

which is rejected by S_R. This terminates the counterexample analysis because the algorithm has found the lowest return index $r_l = 7$ such that $[\![ce[,r_l]]\!]^* \cdot ce[r_l+1,] \in L(S_R)$. The violating procedure (P^G) is identified by the matching call symbol and $\widehat{c} \cdot \widehat{a} \cdot \widehat{F}$ is extracted as the (projected) local counterexample. Line 18 delegates this local counterexample to the respective procedural learner for the refinement of P^G. The result of this process (and consequently, the updated SPA hypothesis H^4_{SPA}) is shown in Figure 6.3d.

Further Counterexamples The learning process continues with receiving counterexamples that further expose in-equivalences in the procedures of the current SPA hypothesis. Due to the proposed counterexample analysis (cf. Corollary 3) and its implementation using the alpha-gamma transformation with its properties of monotonicity (cf. Theorem 9), the global counterexamples continue to be projected to local counterexamples of the involved procedures so that the soundness of the respective procedural learners guarantees that the final SPA hypothesis eventually coincides with the SUL shown in Figure 3.1.

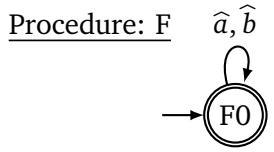
6.2.4 Termination and Complexity

For discussing the properties of termination and complexity of the SPA learner, recall that in the MAT framework, a learner has access to MQs for exploring the behavior of the SUL and equivalence queries (EQs) to verify the intermediate hypotheses. The following analysis assumes that the SUL is represented by a minimal SPA of size n such that $|P^{c_i}| = n_i$ for all $c_i \in \Sigma_{call}$. Under the assumption that one uses one of the well-known regular learning algorithms which are able of infer canonical procedural hypotheses of P^{c_i} with a maximum of n_i EQs, the following holds.

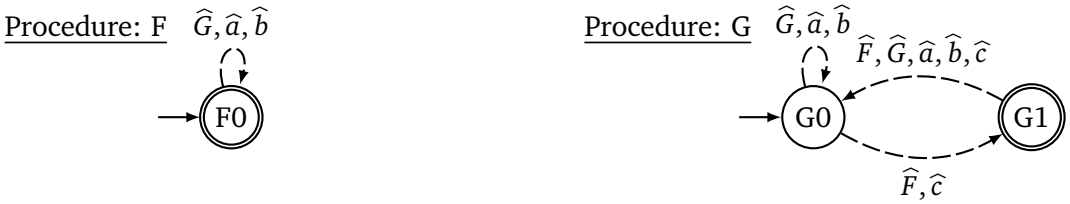
Theorem 10 (Correctness and termination [61])
Let Σ be an SPA input alphabet and S^{SUL} be a minimal SPA over Σ with size n. Having access to a MAT for $L(S^{SUL})$, the SPA learning algorithm infers a minimal SPA model S^{mod} with $S^{mod} \equiv_{SPA} S^{SUL}$ requiring at most $n+1$ EQs.

Proof. This property is based on the fact that every valid global counterexample detects an inconsistency in one of the procedures of H_{SPA} and therefore allows one to extract a local counterexample. Every local counterexample increases the number of states of the concerned procedures until they are equivalent to their SUL counterpart and therefore no more global counterexamples can expose any inconsistencies in the concerned procedures. Since every procedural hypothesis requires a maximum of n_i local counterexamples before equivalence is achieved and one additional EQ is required to indicate equivalence, the bound directly follows. For the full proof, see Theorem 4 in [61]. □

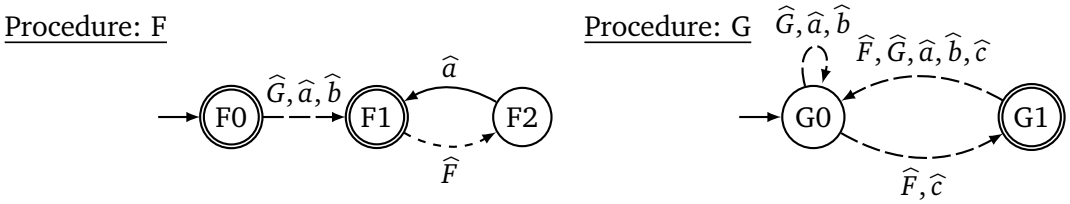
Figure 6.3
The procedural hypothesis models of H_{SPA} after the respective counterexample refinements. Sink states and corresponding transitions are omitted for readability.



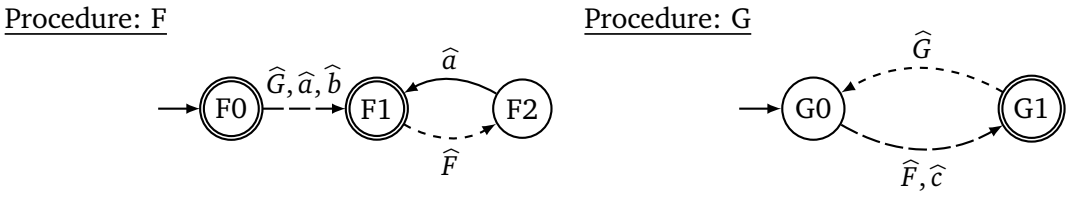
(a) The procedural hypothesis models of H^1_{SPA} after the first counterexample refinement.



(b) The procedural hypothesis models of H^2_{SPA} after the second counterexample refinement.



(c) The procedural hypothesis models of H^3_{SPA} after the third counterexample refinement.



(d) The procedural hypothesis models of H^4_{SPA} after the fourth counterexample refinement.

Regarding query complexity, the notion of rigorous (de-) composition results in a query complexity of the SPA learner that is determined by the aggregation of the individual query complexities of the regular learning algorithms of the involved procedures.

Theorem 11 (Query complexity [61])
Let Σ be an SPA input alphabet and S^{SUL} be a minimal SPA over Σ with size n and k procedures. Let C_{c_i} denote the query complexity of the local learner for inferring procedure P^{c_i} with $c_i \in \Sigma_{call}$, and let m denote the length of the longest (global) counterexample. Having access to a MAT for $L(S^{SUL})$, inferring a minimal SPA model S^{mod} such that $S^{mod} \equiv_{SPA} S^{SUL}$ has a query complexity of $\mathcal{O}\left(\left(\sum_{i=1}^{k} C_{c_i}\right) + n \log_2 m\right)$.

Proof. This directly follows from the compositional nature of the SPA inference process: Procedural hypotheses are constructed by their corresponding procedural learners and therefore require the respective amount of membership queries. In addition to that, the global counterexample analysis can be implemented in a binary search fashion and therefore only adds a logarithmic term to the query performance. For the full proof, see Theorem 5 in [61]. □

6.2.5 Optimization Heuristics

As Theorem 11 shows, the query complexity of the SPA learner is mainly determined by the query complexity of the involved procedural learners. This means that any performance improvements in the field of regular AAL can be directly transferred to the learning process of (instrumented) context-free systems. This not only affects query complexity but also *symbol complexity*, i.e., the number of symbols occurring in the posed MQs. Another major factor that affects this metric of the SPA learner is the gamma expansion, specifically the length of the used access sequences, terminating sequences, and return sequences.

In the following, we look at the exchangeability of terminating sequences and (pairs of matching) access sequences and terminating sequences between two independent query expansions. This opens up the way to optimizing, i.e., shortening, these sequences throughout the learning process and establishing a notion of automated self-optimization during learning. Recall from Definition 35 that there exist no special structural restrictions on the three types of sequences as long as the combined sequences represent an accepted word of an SPA. The two proposed optimization heuristics are based on the ideas of *terminating sequence invariance* and *context invariance*.

Theorem 12 (Terminating sequence invariance)
Let Σ be an SPA input alphabet and S be an SPA over Σ.

$$(\widehat{c} \cdot \widehat{v}, \sigma)_{SPA} \xrightarrow{c \cdot ts_c \cdot r}{}^{*} (\widehat{v}, \sigma)_{SPA}$$

for all $c \in \Sigma_{call}, ts_c \in TS_c$ and some matching $\widehat{v} \in \widehat{\Sigma}^{*}_{proc}$.

Proof. This directly follows from Definitions 29 and 35. Since terminating sequences are extracted from accepted words of an SPA, we can choose the respective w of Definition 35 such that we have $w[i] = c, w[i+1, \rho_w(i+1)] = ts_c, w[\rho_w(i+1)+1] = r$ for some

matching $\langle c, i\rangle \in Inst_w$. Since the respective w is an accepted word of S, there exists an admissible path in the language-SOS system that emits $c \cdot ts_c \cdot r$ after emitting the access sequence $w[,i-1]$. By Definition 29 c can only be emitted if $\widehat{c}$ is the first alphabet symbol of the state of the current SOS configuration and the well-matchedness of ts_c ensures that we reach the same σ after emitting the matching return symbol. This directly concludes the statement. □

Theorem 13 (Context invariance)
Let Σ be an SPA input alphabet, S be an SPA over Σ, $c \in \Sigma_{call}$, and $ts_c \in TS_c$.

$$as_c \cdot ts_c \cdot rs_c \in L(S)$$

for all $\langle as_c, rs_c\rangle \in Cont_c$.

Proof. Let $c \in \Sigma_{call}$, $\langle as_c, rs_c\rangle \in Cont_c$ be arbitrary and $ts_c \in TS_c$ be fixed. We have to show that $as_c \cdot ts_c \cdot rs_c \in L(S)$ or equivalently

$$(\widehat{c}_0, \bot)_{SPA} \xrightarrow{as_c \cdot ts_c \cdot rs_c}{}^{*} (\varepsilon, \bot)_{SPA}.$$

Since $\langle as_c, rs_c\rangle \in Cont_c$, there exists by Definition 35 a $ts'_c \in TS_c$ such that $as_c \cdot ts'_c \cdot rs_c \in L(S)$. By Definition 29 we have

$$\begin{aligned}
(\widehat{c}_0, \bot)_{SPA} &\xrightarrow{as_c[,|as_c|-1]}{}^{*} (\widehat{c} \cdot \widehat{w}_1, \sigma)_{SPA} \\
&\xrightarrow{c} (\widehat{w}_2 \cdot \widehat{r}, \widehat{w}_1 \bullet \sigma)_{SPA} \\
&\xrightarrow{ts'_c}{}^{*} (\widehat{r}, \widehat{w}_1 \bullet \sigma)_{SPA} \\
&\xrightarrow{r} (\widehat{w}_1, \sigma)_{SPA} \\
&\xrightarrow{rs_c[2,]}{}^{*} (\varepsilon, \bot)_{SPA}
\end{aligned}$$

for some $\widehat{w}_1, \widehat{w}_2 \in \widehat{\Sigma}^*_{proc}, \sigma \in ST(\Gamma_{SPA})$. Since $ts_c \in TS_c$, we know by Theorem 12 that

$$(\widehat{c} \cdot \widehat{w}_1, \sigma) \xrightarrow{c \cdot ts_c \cdot r} (\widehat{w}_1, \sigma)$$

which allows us to replace ts'_c with ts_c, i.e., there exists a $\widehat{w}_3 \in L(P^c)$ such that

$$(\widehat{c}_0, \bot)_{SPA} \xrightarrow{as_c}{}^{*} (\widehat{w}_3 \cdot \widehat{r}, \widehat{w}_1 \bullet \sigma)_{SPA} \xrightarrow{ts_c}{}^{*} (\widehat{r}, \widehat{w}_1 \bullet \sigma)_{SPA} \xrightarrow{rs_c}{}^{*} (\varepsilon, \bot)_{SPA}$$

which concludes the statement. □

Theorems 12 and 13 state that one can arbitrarily exchange terminating sequences and contexts (pairs of matching access sequences and return sequences) without affecting the membership property of the concerned word. As a result, the gamma expansion may choose arbitrary representatives when expanding local queries to global queries without affecting the correctness of the answers to procedural MQs. This is a powerful enabler for reducing the effective symbol complexity of the learning process in practice. Rather than restricting the analysis of positive counterexamples to extracting the initial terminating

sequences and (pairs of matching) access sequences and return sequences only, the SPA learner may scan *every* positive counterexample to see if it contains shorter instances of the respective sequences. If it finds shorter sequences, the leaner can *replace* the existing representatives and therefore construct shorter expanded queries afterwards. This concept adds a sense of automated self-improvement to the learning process.

Especially for terminating sequences, this concept can be applied recursively. Access sequences, terminating sequences, and return sequences from positive counterexamples may contain nested procedural calls themselves. Theorem 12 allows one to replace these invocations with arbitrary, i.e., shortest, terminating sequences *as well*. For example, this means that a positive counterexample that initially improves the terminating sequence of a procedure c may also improve the access sequence of a procedure c' if it contains nested calls to c. By repeatedly, e.g., for every positive counterexample, analyzing the existing sequences for possible replacements of nested procedural invocations with new terminating sequences, the learner may further reduce the symbol complexity of the expansion mechanism.

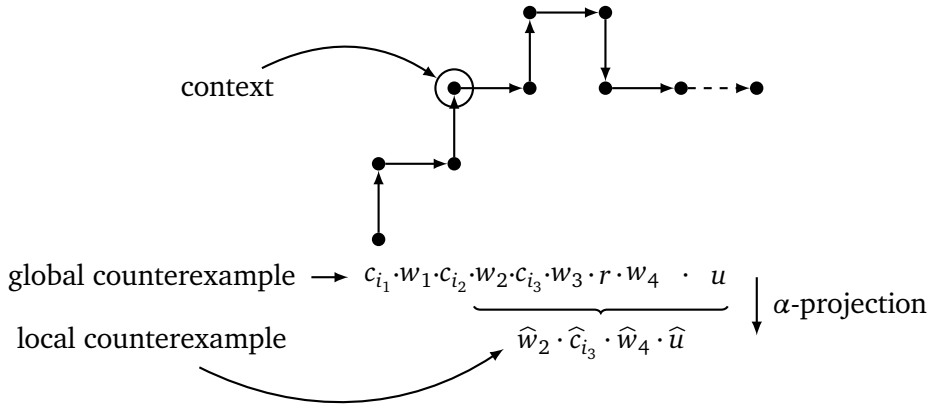
Depending on the chosen procedural learners, there may exist another source for finding short(er) terminating sequences besides positive counterexamples. Some (regular) AAL algorithms represent states of their current hypothesis via representatives $\widehat{w} \in \widehat{\Sigma}^*_{proc}$. If a state is accepting, the corresponding representative constitutes a successful run of the procedure. This representative (possibly gamma-expanded if it contains nested calls) is another candidate for a shorter terminating sequence. For example, if the initial state of a procedural hypothesis is accepting, ε is a valid (shortest) terminating sequence for this procedure, irrespective of what other terminating sequences are extracted from positive counterexamples. This search for terminating sequences can be performed for all procedural learners independently and does not require any additional interaction with the SUL as the representatives can be extracted from the hypotheses directly.

By constantly analyzing positive counterexamples and the current hypotheses of the procedural learners (if possible), one can repeatedly look for shorter access sequences, terminating sequences, and return sequences in order to continuously improve the symbol performance of the SPA learner throughout the learning process. Section 10.2.2 shows the impact of these optimizations by comparing setups that use the previously mentioned steps and setups that simply use the initial sequences obtained from positive counterexamples.

6.3 SBAs

The SBA learner needs to address the semantic characteristics of SBAs, namely the prefix-closure of both the global language and the individual procedural languages. Compared to the SPA case, there are two aspects of prefix-closure that allow one to simplify the learning process and two aspects that require additional adjustments of the learning process. We continue with the simplifications first and then look at the necessary adjustments.

Figure 6.4
Extraction of a local counterexample from a reduced global counterexample ce. The context of the in-equivalent action u can be determined by the highest index i_* such that $\beta(ce[i_*, |ce|-1]) > 0$.



6.3.1 Simplifications

The first simplification concerns the expansion of local queries during the exploration phase. Theorem 5 shows that for answering procedural MQs on the global SUL, it is sufficient to concatenate an access sequence with the specifically expanded procedural query, i.e., there is no need for return sequences anymore. Consequently, the discovery and management of these sequences can be removed from the learning process.

The second simplification concerns the analysis of counterexamples. For regular SPAs, an intricate analysis for determining a violating procedure is required which may involve executing additional queries on the SUL (cf. Section 6.2.2). For prefix-closed languages, this process can be simplified by establishing the notion of *reduced counterexamples* which expose an inconsistency between the current hypothesis and the SUL at the last symbol of the counterexample.

Definition 67 (Reduced counterexample)
Let Σ be an SPA input alphabet, $ce \in \Sigma^*$ denote a counterexample, and S_A, S_R denote two SPAs over Σ such that $ce \in L(S_A)$ and $ce \notin L(S_R)$. We call ce reduced *iff $ce[, |ce|-1] \in L(S_R)$.*

As shown in Figure 6.4, a reduced counterexample directly allows one to identify the procedure that causes the in-equivalent behavior by determining the context in which $ce[|ce|]$ is emitted. The local counterexample can be extracted by projecting the suffix up to the corresponding call symbol without requiring an intricate analysis process involving the alpha-gamma transformation anymore. This also makes any related checks such as the ts-conformance of the current hypothesis model obsolete. All this is a direct consequence of Theorem 4.

The additional requirement (towards the EQO) to provide reduced counterexamples only needs minor adjustments. Any test for equivalence that the EQO wants to perform

only needs to be evaluated in a step-wise fashion such that the (global) counterexample is returned after the first observed mismatch in behavior. Given that SBAs naturally represent reactive systems, this step-wise evaluation is easily achievable. For SULs that do not support this type of interaction, a reduced counterexample can still be constructed by testing multiple prefixes of the initial counterexample. Since the index of the first mismatching symbol can be determined via a binary search, this approach performs no worse than the SPA case.

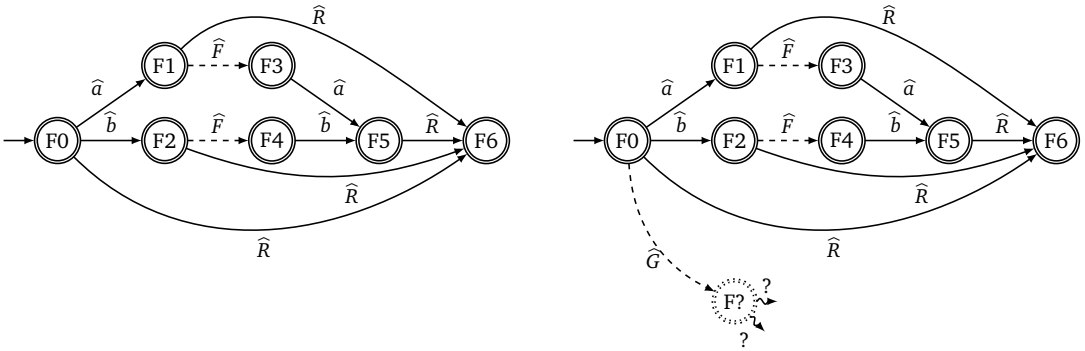
6.3.2 Adjustments

The first adjustment addresses the fact that behavioral automata include the return symbol in their procedural alphabet. As a result, procedural MQs may now also include the return symbol. More critically, procedural MQs may contain symbols *beyond* the return symbol. Similar to the situation during conformance testing (cf. Section 5.3.2), the gamma expansion cannot properly expand these queries in order to evaluate them on the SUL. Again, the notion of return-closure of behavioral automata comes as a remedy. As these queries are rejected by valid behavioral automata, the procedural MQOs can short-circuit these queries, i.e., upon detecting that a local query extends beyond a return symbol, the query is neither expanded nor delegated to the SUL but answered with *false* immediately. By allowing only the last symbol of a local query to be the return symbol and expanding queries similar to Theorem 5, one can circumvent any problems regarding expansion without affecting the correctness of the answers to the queries.

Besides query expansion, including the return symbol in the procedural alphabet may also affect the validity of intermediate hypotheses. If the procedural learner does not (yet) investigate successors of return transitions, it may happen that intermediate procedural hypotheses violate the required return-closure of behavioral automata. For example, consider an initial hypothesis with only a single accepting state in which all transitions loop back into the initial state. This inconsistency can be easily addressed via a structural analysis which verifies that all return successors lead into a rejecting sink state. If a procedural hypothesis does not satisfy this property, paths starting at the initial state and ending with the respective return transitions can be used as negative counterexamples to refine the hypothesis. By repeatedly checking this property, e.g., after each refinement, the SBA learner can assure the return-closure of the current procedural hypotheses. Note that these operations, i.e., accessing a hypothesis model and supplying a counterexample, are all standard operations for (regular) AAL algorithms. As a result, one can still use any MAT-compatible learner for SBA learning.

The second adjustment concerns the process of incorporating information from counterexamples during the verification phase. For SPAs (cf. Section 6.2.1), positive counterexamples provide access sequences, terminating sequences, and return sequences that allow the SPA learner to extend the learning alphabet of procedural learners and progress the exploration of procedural hypotheses. While the SBA learner no longer needs return sequences for query expansion, access sequences and terminating sequences are still very much required.

Figure 6.5 (from [62])
A hypothesis of P_B^F of Figure 3.5 without any procedural G transitions (left) and after incorporating the information of a positive counterexample "$F \cdot G$" (right).



However, with prefix-closed languages, a positive counterexample may no longer provide these two sequences simultaneously. Consider a potential positive counterexample $F \cdot G$ for the SBA based on Figure 3.5. While it provides access sequences for both procedures F and G, it does not provide terminating sequences for any of the two procedures. Therefore, the SBA learner may activate the procedural learners of F and G but it cannot add the two call symbols to the set of active learning symbols. However, at the same time, the hypothesis of procedure F needs to reflect the behavior of G, i.e., the successful invocation of G. Otherwise, the learning process potentially deadlocks with a valid counterexample but no hypothesis refinement.

The problem with the above type of counterexamples is that it introduces *divergent states*, i.e., states for which future behavior cannot be determined yet. Figure 6.5 visualizes this problem. The divergent state may at some point have valid accepting successors. For example, if the SBA learners receives the positive counterexample "$F \cdot G \cdot c \cdot R \cdot R$", it can extract c as a terminating sequence for G and the respective procedural MQOs can properly expand calls to G. However, in an instance where the learning process terminates without such a counterexample, all transitions of the divergent state must lead into a sink state because procedure G is found to be non-terminating.

Divergent states pose new challenges for the AAL process of SBAs because they may not necessarily represent actual states in the hypothesis. In case of Figure 6.5, the divergent state F? later coincides with the state F5. As a result, the hypothesis construction is no longer guaranteed to be a monotonic process which is a fundamental requirement for a lot of termination and correctness proofs of learning algorithms.

The SBA learner may tackle this problem by re-using the idea of *incremental alphabet extension*. When a new call symbol is first encountered in a positive counterexample and no terminating sequence can be extracted, the learner may extend the active learning alphabet with a *non-terminating version* of the call symbol. When the procedural MQOs are given local queries that continue beyond non-terminating call symbols, they short-circuit

these queries and directly answer them with *false* without delegating the query to the SUL, similar to queries that extend beyond the return symbol. As a result, the divergent states are forced to materialize as non-terminating states, i.e., all its successors lead into the sink state. Note that local queries that end with non-terminating call symbols, e.g., "$\widehat{G}$" in case of P_B^F of Figure 6.5, do not need their last symbol expanded (cf. Theorem 5) and can still be answered correctly by the (prefix-closed) SUL. This means that the information of a positive counterexample, such as $F \cdot G$, can be correctly incorporated into the tentative hypothesis of procedure F. If the learner at some point later receives a global counterexample that provides a terminating sequence, such as "$F \cdot G \cdot c \cdot R \cdot R$", it adds a *terminating version* of the call symbol to the active learning alphabet, which is treated similar to the SPA case.

This means that the procedural hypotheses of SBAs are generally defined over $\widehat{\Sigma} \uplus \widehat{\Sigma}'_{call}$, where in this case $\widehat{\Sigma}'_{call}$ denotes the set of non-terminating call symbols. In order to decide which version is the correct one, the learner simply maintains a mapping $\Sigma_{call} \to \widehat{\Sigma}_{call} \uplus \widehat{\Sigma}'_{call}$ throughout the learning process that decides for each call symbol if the non-terminating or terminating version should represent the behavior of the hypothesis, depending on whether it has found a terminating sequence or not. Using this mapping, the learner can always provide a unique and deterministic view on the procedural hypotheses that reflects the current knowledge of the learning process.

Note that if the SBA learner directly receives a positive counterexample that provides a terminating sequence, e.g., "$F \cdot G \cdot c \cdot R \cdot R$" instead of "$F \cdot G$" in the case Figure 6.5, no non-terminating call symbol needs to be introduced and the learning process may proceed directly like in the case of SPAs.

6.3.3 Termination and Complexity

Given that the approach for tackling divergent states is (at least in this thesis) implemented via incremental alphabet extension and (regular) AAL algorithms behave monotonically with respect to additional alphabet symbols, the general argumentation for termination and correctness known from SPAs (cf. Section 6.2.4) directly applies to SBAs as well.

However, regarding query complexity, the previously discussed simplifications and adjustments change the dimension of this bound. On the one hand, the complexity of procedural learners often depends on the size of the procedural hypothesis and the number of its input symbols. By dealing with divergent states via additional input symbols, the size of the procedural alphabet increases up to $|\widehat{\Sigma}| + |\widehat{\Sigma}_{call}|$ (compared to $|\Sigma_{proc}|$ for SPAs). On the other hand, reduced global counterexamples do not require any analysis. This leads to the following query complexity of the SBA learner.

Theorem 14 (Query complexity)
Let Σ be an SPA input alphabet and S_B^{SUL} be a minimal SBA over Σ with k procedures. Let C_{c_i} denote the query complexity of the local learner for inferring procedure $P_B^{c_i}$ with $c_i \in \Sigma_{call}$. Having access to a MAT for $L(S_B^{SUL})$ that returns reduced counterexamples, inferring a minimal SBA model S_B^{mod} such that $S_B^{mod} \equiv_{SBA} S_B^{SUL}$ has a query complexity of $\mathcal{O}\left(\left(\sum_{i=1}^{k} C_{c_i}\right)\right)$.

Proof. Similar to Theorem 11, this directly follows from the compositional nature of the SBA inference process: Procedural hypotheses are constructed by their corresponding procedural learners and therefore require the respective amount of membership queries. Analyzing reduced counterexamples does not cause any additional queries. □

6.4 SPMMs

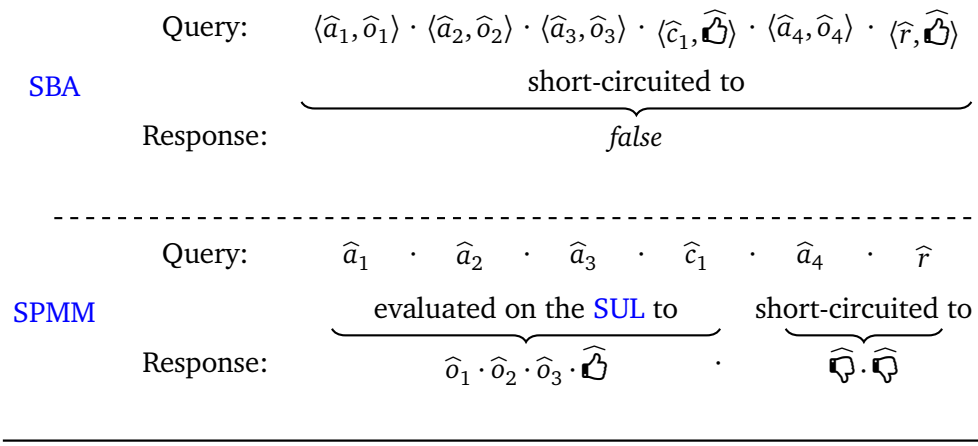
Formally, the semantics of SPMMs are defined via SBAs over the cartesian product of some SPA input alphabet and some SPA output alphabet. Consequently, the results of Section 6.3 directly lead to a possible implementation of an SPMM learner. Therefore, this section primarily focuses on possible advantages that native Mealy-based representations of procedures offer in the context of AAL. For example, consider the SPMM based on Figure 3.7 and its respective SBA-based characterization, i.e., an output-enriched version of the SBA based on Figure 3.5. For the procedural learner of P_B^F to infer the transduction $\langle a, x \rangle$ it would need to pose the procedural MQs $\langle \widehat{a}, \widehat{x} \rangle, \langle \widehat{a}, \widehat{y} \rangle$, and $\langle \widehat{a}, \widehat{z} \rangle$ only to have solely the query $\langle \widehat{a}, \widehat{x} \rangle$ get answered with *true*. Here, the learner requires three queries to determine a single transduction step.

With a native transduction-based SUL that operates in a deterministic and incremental lock-step-based pattern similar to SPMMs, the learner (MQO, respectively) simply poses the query $\widehat{a}$ so that the system answers it with $\widehat{x}$, requiring only a single query for the same amount of information. Many (regular) AAL algorithms have been extended to support these kinds of MQs in order to infer native Mealy machines (cf. Section 8.3). Therefore, by implementing a native SPMM learner that is provided with procedural learners which use a Mealy-specific MQO, the performance of the learning process can be improved by requiring fewer queries.

While many of the ideas of Section 6.3, such as the handling of divergent states via incremental alphabet extension, directly apply to the SPMM case as well, there is a slight adjustment needed for short-circuiting queries. As discussed in Section 6.3.2, handling procedural MQs that contain symbols beyond non-terminating call symbols or the return symbol are short-circuited to be rejected automatically. In lock-step-based transductions, the number of symbols of an output word must match the number of symbols of the input word, even if the query needs to be short-circuited because it contains non-continuable input symbols. This requirement can be easily met by first evaluating the query up until the first non-continuable input symbol on the SUL. Due to the prefix-closure of the considered transductions, the SUL still answers all inputs up until this symbol correctly. For the remainder of the query, the procedural MQO then simply *repeats* the "error" output symbol 👎 as this symbol represents the rejecting behavior of behavioral automata. This small adjustments allows the procedural MQOs to transfer the SBA-specific semantics seamlessly to the SPMM case. Figure 6.6 sketches this process for an exemplary query.

Compared to SBAs, SPMM-based transductions furthermore allow for an easier construction of reduced counterexamples. Since the output *words* essentially record all intermediate responses of the system, the symbol (index) at which the behavior of the SUL and the behavior of the current SPMM hypothesis differ is easily determined by a

Figure 6.6
A comparison between answering SBA queries and SPMM queries with a non-terminating call symbol c_1.



symbol-wise comparison of the output symbols. Cutting off the counterexample after the first mismatch of output symbols directly yields the reduced counterexample required for identifying the violating procedure.

6.5 Summary

This section concludes the chapter by summarizing its main results.

- Similar to the field of model-based testing (MBT), the notion of rigorous (de-) composition of SPAs, SBAs and SPMMs allows one to implement the inference of the global system as a simultaneous inference of the local procedures.
- For SPAs,
  - the exploration phase is a straightforward implementation of Theorem 2 in which the inference of individual procedures is deferred until the learner has all necessary information about access sequences, terminating sequences, and return sequences available.
  - the verification phase involves a two-step process, in which (first) the global counterexample for the SPA is analyzed to identify a violating procedure via an efficient binary search so that (second) the actual refinement is delegated to the corresponding procedural learner by constructing a projected (local) counterexample according to Theorem 1.
  - the invariance of terminating sequences and contexts (pairs of access sequences and return sequences) offers one the possibility to replace these sequences *during* the learning process, allowing for an automated optimization of the query expansion process.
- For SBAs,

  - the inference process simplifies because the prefix-closure of SBA languages no longer requires the management of return sequences. Furthermore, the semantics of SBAs allow for the notion of reduced counterexamples in which the violating procedure can be determined directly without requiring additional analysis like in the SPA case.
  - the inference process requires adjustments because access sequences and terminating sequences may no longer occur simultaneously. In order to represent non-terminating procedures (or procedures for which terminating sequences are unknown at the time), the proposed SBA learner distinguishes between terminating call symbols and non-terminating call symbols, in order to guarantee a sound progression using incremental alphabet extension.
  - the otherwise similar concepts to the case of SPAs allow one to exploit the previously established properties such as sequence invariance.
- For SPMMs, the direct response mechanism of lock-step pattern-based SULs allows one to implement more efficient procedural MQs compared to the SBA case.
- For all formalisms, the required steps (expansion, projection, refinement, etc.) do not require any modifications to (regular) AAL algorithms. As a result, the proposed SPA, SBA, and SPMM learners may be parameterized by any MAT-compatible (regular) learners for the inference of the respective procedures.

CHAPTER 7

Transformations Between SPAs and VPAs

This chapter discusses the formalism of visibly push-down automata (VPAs) by Alur et al. [11] and their relationship with systems of procedural automata (SPAs), systems of behavioral automata (SBAs), and systems of procedural Mealy machines (SPMMs). It compares the expressiveness of VPAs with the expressiveness of SPAs and presents potential transformations between the two formalisms.

7.1 Visibly Push-Down Automata

One of the essential concepts of SPAs, SBAs, and SPMMs is an instrumentation that makes calls to and returns from procedures observable. This concept is similar to visibly push-down languages (VPLs) by Alur et al. [11]. VPLs are defined over a (visibly push-down) alphabet which partitions its symbols into *call symbols*, *internal symbols* and (multiple) *return symbols*. VPLs are accepted by VPAs which are classic push-down automata whose stack operations are determined solely by parsing call symbols and return symbols, hence the term *visibly* push-down automata. The stack represents a *call stack* and keeps track of the current nesting of call symbols.

To allow for a closer comparison between SPAs and VPAs, we first look at the notion of a visibly push-down alphabet and then the formal definition of VPAs.

Definition 68 (Visibly push-down alphabet [11])
A visibly push-down alphabet *is a disjoint union* $\widetilde{\Sigma} = \widetilde{\Sigma}_{call} \uplus \widetilde{\Sigma}_{int} \uplus \widetilde{\Sigma}_{ret}$ *where*

- $\widetilde{\Sigma}_{call}$ *denotes the finite* call alphabet,
- $\widetilde{\Sigma}_{int}$ *denotes the finite* internal alphabet, *and*
- $\widetilde{\Sigma}_{ret}$ *denotes the finite* return alphabet.

Similar to SPAs, the following uses a special markup token $\widetilde{}$ to denote when input symbols, words, or alphabets are interpreted in a VPA context.

Definition 69 (VPA [11])
Let $\widetilde{\Sigma}$ *be a visibly push-down alphabet. A* VPA *over* $\widetilde{\Sigma}$ *is a tuple* $V = \langle Q, Q_{in}, \Gamma, \delta, Q_F \rangle$ *where*

- Q *is a finite set of locations,*
- $Q_{in} \subseteq Q$ *is a set of initial locations,*
- Γ *is a finite stack alphabet including a special bottom-of-stack symbol* $\bot$*,*
- $\delta = \delta_{call} \uplus \delta_{int} \uplus \delta_{ret}$ *is the transition relation where*

  - $\delta_{call} \subseteq (Q \times \widetilde{\Sigma}_{call} \times Q \times (\Gamma \setminus \{\bot\}))$ *is the call transition relation,*
  - $\delta_{int} \subseteq (Q \times \widetilde{\Sigma}_{int} \times Q)$ *is the internal transition relation,*
  - $\delta_{ret} \subseteq (Q \times \widetilde{\Sigma}_{ret} \times \Gamma \times Q)$ *is the return transition relation, and*
- $Q_F \subseteq Q$ *is a set of accepting locations.*

Note that while the above definition allows for non-deterministic VPAs, non-deterministic VPAs and deterministic VPAs are equally expressive [11]. Therefore, the following sections assume all VPAs to be deterministic for reasons of simplicity. Furthermore, the following sections assume all VPAs to be total, i.e., the transition relations describe functions that are defined for all possible input parameters. Similar to regular automata, VPAs can be made total by introducing a sink location and adding transitions to that sink location for all previously undefined transitions.

7.1.1 Semantics

A *run* of a VPA induces a transition system where each state is characterized by a location $q \in Q$ and a stack configuration σ over the domain $\Gamma_{VPA} = (\Gamma \setminus \{\bot\})^* \uplus \{\bot\}$. Again, $\bullet$ is used to denote the stacking of elements where the display of elements from left to right denotes the stack contents from top to bottom. $ST(\Gamma_{VPA})$ denotes the set of all possible *stack configurations*. Definition 70 then formally defines a run of a VPA.

Definition 70 (Run of a VPA [11])
Let $\widetilde{\Sigma}$ denote a visibly push-down alphabet and V denote a VPA over $\widetilde{\Sigma}$. Let $\widetilde{w} = \widetilde{a}_1 \cdot \ldots \cdot \widetilde{a}_k \in \widetilde{\Sigma}^*$ denote a word over $\widetilde{\Sigma}$. The $\widetilde{w}$-induced run of V *is a sequence $\tau = \langle q_0, \sigma_0 \rangle, \ldots, \langle q_k, \sigma_k \rangle$ such that*

- $q_i \in Q$ *for all* $i \in \{0, \ldots, k\}$,
- $\sigma_i \in ST(\Gamma_{VPA})$ *for all* $i \in \{0, \ldots, k\}$,
- $q_0 \in Q_{in}$,
- $\sigma_0 = \bot$,
- $\forall i \in \{1, \ldots, k\}$ *the following holds:*
  - *if* $\widetilde{a}_i \in \widetilde{\Sigma}_{call}$, *then* $\exists \gamma \in \Gamma$ *such that* $\langle q_i, \widetilde{a}_i, q_{i+1}, \gamma \rangle \in \delta_{call}$ *and* $\sigma_{i+1} = \gamma \bullet \sigma_i$,
  - *if* $\widetilde{a}_i \in \widetilde{\Sigma}_{int}$, *then* $\langle q_i, \widetilde{a}_i, q_{i+1} \rangle \in \delta_{int}$ *and* $\sigma_{i+1} = \sigma_i$,
  - *if* $\widetilde{a}_i \in \widetilde{\Sigma}_{ret}$, *then* $\exists \gamma \in \Gamma$ *such that* $\langle q_i, \widetilde{a}_i, \gamma, q_{i+1} \rangle \in \delta_{ret}$ *and* $\gamma \neq \bot \wedge \sigma_i = \gamma \bullet \sigma_{i+1}$ *or* $\gamma = \sigma_i = \sigma_{i+1} = \bot$.

We call a run accepting *iff $q_k \in Q_F$. The* language *of a VPA is defined as the set of all words whose induced runs are accepting, i.e.,*

$$L(V) = \{\widetilde{w} \in \widetilde{\Sigma}^* \mid \textit{the } \widetilde{w}\textit{-induced run is accepted by } V\}.$$

Remark 5
Note that Definition 70 is cited from [11]. However, the definition contains off-by-one errors on the indices of the locations used in the constraints on the transition relation. Instead, it should read "if $\widetilde{a}_i \in \widetilde{\Sigma}_{call}$, then $\exists \gamma \in \Gamma$ such that $\langle q_{i-1}, \widetilde{a}_i, q_i, \gamma \rangle \in \delta_{call}$ and $\sigma_i = \gamma \bullet \sigma_{i-1}$", et cetera.

This directly leads to the definition of VPLs.

Definition 71 (VPL [11])
Let $\widetilde{\Sigma}$ be a visibly push-down alphabet. A language $L \subseteq \widetilde{\Sigma}^*$ is a called a VPL iff there exists a VPA V such that $L = L(V)$.

Note that the acceptance of a word only depends on reaching an accepting location, i.e., there are no requirements on the structure of the stack. This means that VPAs can accept return-matched words, i.e., words with un-matched call symbols (non-empty stack), and call-matched words, i.e., words with un-matched return symbols (return transitions with empty stack), as well. As a consequence, VPLs are (in general) strictly more expressive than SPA languages and SBA languages.

For the comparison between VPAs and SPAs, this thesis restricts itself to VPAs that only accept well-matched languages. It can be argued that this restriction does not impact many practical scenarios. In general, VPAs (thus VPLs) do not support a canonical form which is a fundamental requirement for many techniques from the fields of model-based testing (MBT) and active automata learning (AAL). Alur et al. [12] propose a special form of VPAs, called k-module single-entry visibly push-down automata (SEVPAs), which support canonicity and are limited to describe well-matched VPLs only. Therefore, the remaining discussion about transformations between SPAs and VPAs focuses on analyzing the equivalence of SPAs and SEVPAs. Section 7.4 briefly looks at concepts such as behaviors (SBAs) and transductions (SPMMs) in the context of VPAs. Furthermore, there exist other, e.g., grammar-based, representations of VPLs [21] but this chapter focuses on the automaton-based one due to its similarity with SPAs, which allows for a certain level of convenience when discussing transformations between the two formalisms. We continue with the presentation of k-SEVPAs and their properties.

7.1.2 Canonicity

In model-based quality assurance (MBQA) (especially in MBT and AAL) it is important to have canonical, i.e., minimally unique (up to isomorphism), models in order to make reasonable statements about the correctness and termination of algorithms. Alur et al. [12] show that, in general, VPAs do not support canonical representations unless one constrains the structure of a VPA. For a canonical representation, Alur et al. [12] introduce the notion of a *k-module single-entry visibly push-down automaton* (k-SEVPA). A k-SEVPA is a VPA in which the locations are partitioned into $k+1$ modules $M_0, \ldots, M_k$ and the call alphabet is partitioned into k classes $C_1, \ldots, C_k$. For $i \in \{1, \ldots, k\}$, every call symbol from a partition class C_i transitions the k-SEVPA into a designated entry location $q_i^* \in M_i$ and the matching return symbols transition the k-SEVPA back into the module in which the call symbol was processed. M_0 is considered the *main* module which contains the initial location(s) and accepting locations. Definition 72 formalizes these concepts.

Definition 72 (k-SEVPA [12])
Let $\widetilde{\Sigma}$ be a visibly push-down alphabet and $\{\widetilde{\Sigma}_{call}^j\}_{j=1}^k$ be a partition of $\widetilde{\Sigma}_{call}$. A VPA $V = \langle Q, Q_{in}, \Gamma, \delta, Q_F \rangle$ is a k-SEVPA with respect to $\{\widetilde{\Sigma}_{call}^j\}_{j=1}^k$ iff there is a partition $\{M_j\}_{j=0}^k$ of Q and distinguished locations $q_j^* \in M_j$ for every $j = 1, \ldots, k$ such that

- $q_0 \in M_0$,
- $\Gamma = \{\bot\} \uplus (Q \times \widetilde{\Sigma}_{call})$,
- *if* $\langle q, \widetilde{c}, q', \langle q, \widetilde{c}\rangle\rangle \in \delta_{call}$ *for some* $\widetilde{c} \in \widetilde{\Sigma}^j_{call}$, *then* $q' = q^*_j$,
- *if* $\langle q, \widetilde{a}, q'\rangle \in \delta_{int}$ *for some* $\widetilde{a} \in \widetilde{\Sigma}_{int}$, *then* $\exists j : q, q' \in M_j$,
- *if* $\langle q', \widetilde{r}, \langle q, \widetilde{c}\rangle, q''\rangle \in \delta_{ret}$ *for some* $\widetilde{c} \in \widetilde{\Sigma}_{call}$, *then* $\exists j : q, q'' \in M_j$,
- $Q_F \subseteq M_0$.

Note that the constraints about the initial state, accepting states, and the transition relations imply that k-SEVPAs can only describe well-matched VPLs. For every call transition from q to q' (pushing a tuple $\langle q, \widetilde{c}\rangle$ onto the stack), the matching return transition (popping the tuple $\langle q, \widetilde{c}\rangle$ from the stack) needs to return to the module of q. Given that the initial location and all accepting locations are restricted to the main module and that there are no return transitions possible with an empty stack, accepted runs must terminate with an empty stack, ensuring the well-matchedness of accepted words.

Furthermore, the stack alphabet is chosen as the most fine-grained stack alphabet possible as every call transition uses the unique combination of the current location and the respective call symbol as the element to push onto the stack. On the one hand, this increases the size of the resulting k-SEVPAs as there need to exist return transitions for all possible stack symbols. On the other hand, it removes a free parameter of k-SEVPAs, which allows for simpler definitions.

The theoretical background to k-SEVPAs is the decomposition of a VPL into $k+1$ congruences such that their respective equivalence classes can be used to construct canonical modules which comprise the k-SEVPA. This is similar to the construction of canonical regular automata based on the equivalence classes of the Nerode congruence [130]. The following discussion uses a slightly adjusted characterization of (well-matched) VPLs proposed by Isberner [93]. It uses only a single congruence which results in the creation of a 1-SEVPA that merges the main module M_0 with the single module M_1, i.e., there exists a single initial location which is also the target location of every call transition. For formally introducing this unified congruence, we first look at the notion of *context pairs*.

Definition 73 (Context Pairs [93])
Let $\widetilde{\Sigma}$ *be a visibly push-down alphabet. The set of* context pairs *over* $\widetilde{\Sigma}$, $CP(\widetilde{\Sigma})$, *is defined as*

$$CP(\widetilde{\Sigma}) = \{\langle \widetilde{u}, \widetilde{v}\rangle \in (WM(\widetilde{\Sigma}) \cdot \widetilde{\Sigma}_{call})^* \times MC(\widetilde{\Sigma}) \mid \beta'(\widetilde{u}) = -\beta'(\widetilde{v})\}$$

where β' *represents the (trivially generalized) call-return balance of Definition 31 that supports multiple return symbols.*

Definition 74 (A unified congruence for well-matched VPLs [93])
Let $\widetilde{\Sigma}$ *be a visibly push-down alphabet and let* L *be a well-matched VPL over* $\widetilde{\Sigma}$. *The relation* $\simeq_L \subseteq WM(\widetilde{\Sigma}) \times WM(\widetilde{\Sigma})$ *is defined via*

$$\widetilde{w} \simeq_L \widetilde{w}' \iff \forall \langle \widetilde{u}, \widetilde{v}\rangle \in CP(\widetilde{\Sigma}) : \widetilde{u} \cdot \widetilde{w} \cdot \widetilde{v} \in L \iff \widetilde{u} \cdot \widetilde{w}' \cdot \widetilde{v} \in L$$

for all $\widetilde{w}, \widetilde{w}' \in WM(\widetilde{\Sigma})$.

The set of context pairs plays an important role for constructing canonical SEVPAs because in analogy to the regular case, they represent "discriminators" that distinguish between the different equivalence classes of a language. However, contrary to the regular case, it is necessary to consider the full syntactic congruence for well-matched VPLs rather than the simpler right-congruence. Given a $\simeq_L$ congruence, the $\simeq_L$-induced 1-SEVPA is constructed as follows:

Definition 75 (A $\simeq_L$-induced 1-SEVPA [93])
Let $\widetilde{\Sigma}$ be a visibly push-down alphabet and L be a well-matched VPL over $\widetilde{\Sigma}$. The $\simeq_L$-induced 1-SEVPA is a tuple $V_{\simeq_L} = \langle Q, Q_{in}, \Gamma, \delta, Q_F \rangle$ where

- $Q = WM(\widetilde{\Sigma})/\simeq_L$,
- $Q_{in} = \{[\varepsilon]_{\simeq_L}\}$,
- $\Gamma = \{\bot\} \uplus (Q \times \widetilde{\Sigma}_{call})$,
- $\delta = \delta_{call} \uplus \delta_{int} \uplus \delta_{ret}$ *where*
  - $\delta_{call} = \{\langle [\widetilde{w}]_{\simeq_L}, \widetilde{c}, [\varepsilon]_{\simeq_L}, \langle [\widetilde{w}]_{\simeq_L}, \widetilde{c} \rangle \rangle \mid \forall \widetilde{w} \in WM(\widetilde{\Sigma}), \widetilde{c} \in \widetilde{\Sigma}_{call}\}$,
  - $\delta_{int} = \{\langle [\widetilde{w}]_{\simeq_L}, \widetilde{a}, [\widetilde{w} \cdot \widetilde{a}]_{\simeq_L} \rangle \mid \forall \widetilde{w} \in WM(\widetilde{\Sigma}), \widetilde{a} \in \widetilde{\Sigma}_{int}\}$,
  - $\delta_{ret} = \{\langle [\widetilde{w}]_{\simeq_L}, \widetilde{r}, \langle [\widetilde{w}']_{\simeq_L}, \widetilde{c} \rangle, [\widetilde{w}' \cdot \widetilde{c} \cdot \widetilde{w} \cdot \widetilde{r}]_{\simeq_L} \rangle \mid \forall \widetilde{w}, \widetilde{w}' \in WM(\widetilde{\Sigma}), \widetilde{r} \in \widetilde{\Sigma}_{ret}, \widetilde{c} \in \widetilde{\Sigma}_{call}\}$,
- $Q_F = \{[\widetilde{w}]_{\simeq_L} \mid \widetilde{w} \in L\}$.

Here, $[\widetilde{w}]$ represents the equivalence class of $\widetilde{w}$ in the quotient set $WM(\widetilde{\Sigma})/\simeq_L$.

The above definition specifically describes the construction of 1-SEVPAs as presented in [93]. However, it is easy to see, how the unified congruence can be generalized to several per-module congruences for any given k-partition of the call alphabet in order to construct a corresponding k-SEVPA [12].

7.2 SPAs as SEVPAs

Isberner [93] shows that in order to construct a 1-SEVPA from a unified congruence $\simeq_L$, one does neither need to consider all possible context pairs nor all possible well-matched words. Instead, a set of *characterizing context pairs* as well as a set of *short prefixes* as representatives of locations suffice to construct a 1-SEVPA describing the language. This section presents how these two sets can be constructed from an SPA S in order to construct a language-equivalent SEVPA by means of Definition 75.

The set of relevant context pairs is specific to a VPL since it needs to properly distinguish between accepted and rejected words of that language. Given an SPA S, its (global) language is characterized by the (local) languages of its involved procedures (cf. Theorem 1) which in turn are characterized by their states. Therefore, by uniquely characterizing each state, it is possible to cover all relevant language characteristics. Given that procedures of S are regular deterministic finite acceptors (DFAs), it is possible to use existing algorithms to compute such characterizing sets (cf. Definition 23) for each procedure. In order to correctly transfer their characterizing properties into the context of well-matched VPLs, it is further necessary to embed the elements of the characterizing sets in their respective context using the corresponding access sequences, terminating sequences, and return

sequences (cf. Theorem 2). The respective sequences can be computed as described in Algorithm 5.1. For a characterizing sequence $\widehat{w}$ of procedure c, the corresponding context pair is then given by $\langle as_c, \gamma(\widehat{w}) \cdot rs_c \rangle$. Note that by definition of access sequences, return sequences, and the gamma expansion, these tuples satisfy the constraints of Definition 73. Furthermore, recall that the membership property of an expanded, procedural word is invariant under valid terminating sequences and pairs of access sequences and return sequences (cf. Theorems 12 and 13). Therefore, any valid instances of the three sequences can be used for the construction of the context pairs.

Additionally, one needs to include a special (base) context pair for separating the initial location, the accepting location, and a (potential) rejecting sink location of the main module because these locations are modeled *externally* via $(\widehat{c}_0, \bot)$ and $(\varepsilon, \bot)$ in Definition 29. Definition 76 summarizes these steps in a formal definition of *SPA-induced context pairs*.

Definition 76 (SPA-induced context pairs)
Let Σ be an SPA input alphabet and S be an SPA over Σ. Let P^c denote a procedural automaton of S for $c \in \Sigma_{call}$. We define the SPA-induced context pairs *as*

$$CP_S(\Sigma) = \left(\bigcup_{c \in \Sigma_{call}} \{ \langle as_c, \gamma(\widehat{w}) \cdot rs_c \rangle \mid \widehat{w} \in CS(P^c) \} \right) \cup \{ \langle \varepsilon, \varepsilon \rangle \}.$$

The combination of Definitions 74 and 76 offers a means for a fix-point-style computation of representatives. Starting with ε as the definitive representative of the initial location, the different cases of Definition 75 for δ_{int} and δ_{ret} suggest new candidates for equivalence classes (representatives, respectively). Note that δ_{call} does not need to be considered explicitly because in the specific case of 1-SEVPAs from [93], all call-transitions must lead to the initial state. By using the previously computed set of SPA-induced context pairs, one can check whether a new candidate, e.g., $\widetilde{w} \cdot \widetilde{a}$, falls into an already discovered equivalence class or represents a new one. In the latter case, $\widetilde{w} \cdot \widetilde{a}$ serves as a representative of the newly discovered equivalence class and the construction of the respective 1-SEVPA according to Definition 75 starts anew. These steps are repeated until a fix-point is reached where no more new equivalence classes can be detected. By evaluating the context pairs on S, i.e., choosing $L = L(S)$ in Definitions 74 and 75, this construction ensures that the resulting 1-SEVPA describes $L(S)$.

Note that due to the finiteness of S, $CP_S(\Sigma)$ only distinguishes between a finite amount of representatives. In the context of Definition 74 this means that the unified congruence $\simeq_{L(S)}$ only has a finite amount of equivalence classes and specifically that each SPA language constitutes a valid well-matched VPL. Given that the set of context pairs is constructed from the (white-box) model of S, this set is complete and correct. Theorem 15 summarizes this result.

Theorem 15
Let Σ be an SPA input alphabet and S be an SPA over Σ. Let $\simeq_{L(S)}$ denote the unified congruence for $L(S)$ according to Definition 74, where $CP(\widetilde{\Sigma})$ is replaced with $CP_S(\Sigma)$ of

Definition 76. Let $V_{\simeq_{L(S)}}$ denote the $\simeq_{L(S)}$-induced 1-SEVPA according to Definition 75 where the representatives of the locations are constructed iteratively on the basis of Definition 76. Then we have

$$L(S) = L(V_{\simeq_{L(S)}}).$$

Proof. This is a direct consequence of Isberner [93] and the construction of $CP_S(\Sigma)$ and $V_{\simeq_{L(S)}}$. The construction of $CP_S(\Sigma)$ on the basis of a (white-box) model of S ensures that $CP_S(\Sigma)$ contains all context pairs for characterizing $L(S)$. By the results of Isberner [93], $\simeq_L$ of Definition 74 only needs a set of characterizing context pairs to fully classify L and the construction of $V_{\simeq_L}$ yields a 1-SEVPA such that $L(V_{\simeq_L}) = L$. Using $L = L(S)$ in the methods of Isberner [93] directly concludes the statement. □

Intuitively, one can think of this process as a white-box version of a VPL-based AAL algorithm. Instead of using counterexamples and the analysis thereof to infer new locations and context pairs that distinguish the locations of a system, the (white-box) model of the SPA serves as a means to directly extract the necessary information and iteratively construct the 1-SEVPA model.

Characterizing a VPL via context pairs and representatives of equivalence classes of its language congruence allows for a flexible framework of transformation, as visualized in Figure 7.1. At the center, there are context pairs and representatives (possibly partitioned into various modules) as the core characterization of a VPL. These may be inferred from various sources such as AAL or the previously discussed analysis of SPAs. Using this core characterization, one is able to construct canonical models in the form of SEVPAs. Depending on the provided partitioning of the respective context pairs and representatives, this allows for k-SEVPAs for arbitrary $k \in \{1, \ldots, |\widetilde{\Sigma}_{call}|\}$. The above SPA-based transformation focuses on a (non-partitioned) 1-SEVPA transformation, but with slight adjustments other partitions can be easily implemented.

7.3 SEVPAs as SPAs

The previous section shows how to construct a SEVPA for representing a given SPA language. In general, the reverse process is not possible. For example, consider the VPL $L = \{ccarcbrr\}$ over the SPA input alphabet $\Sigma = \{c\} \uplus \{a, b\} \uplus \{r\}$. The corresponding SEVPA is shown in Figure 7.2. It is easy to see that this language cannot be represented by an SPA: It would require a single procedural automaton P^c such that for the first invocation of c, it would have to accept the procedural language $L(P^c) = \{\widehat{cc}\}$, for the second invocation of c, it would have to accept the procedural language $L(P^c) = \{\widehat{a}\}$, and for the third and final invocation of c, it would have to accept the procedural language $L(P^c) = \{\widehat{b}\}$.

The main reason why a SEVPA can describe this language but an SPA cannot is the fact that SEVPAs can select individual return successors based on the current location. Depending on whether it reaches a location via a, b, or two invocations of c, it can independently decide with which location the run should continue. This enables SEVPAs to *alias* different procedural behaviors behind a single call symbol. In contrast, SPAs

Figure 7.1
Context pairs and representatives of equivalence classes characterize VPLs.

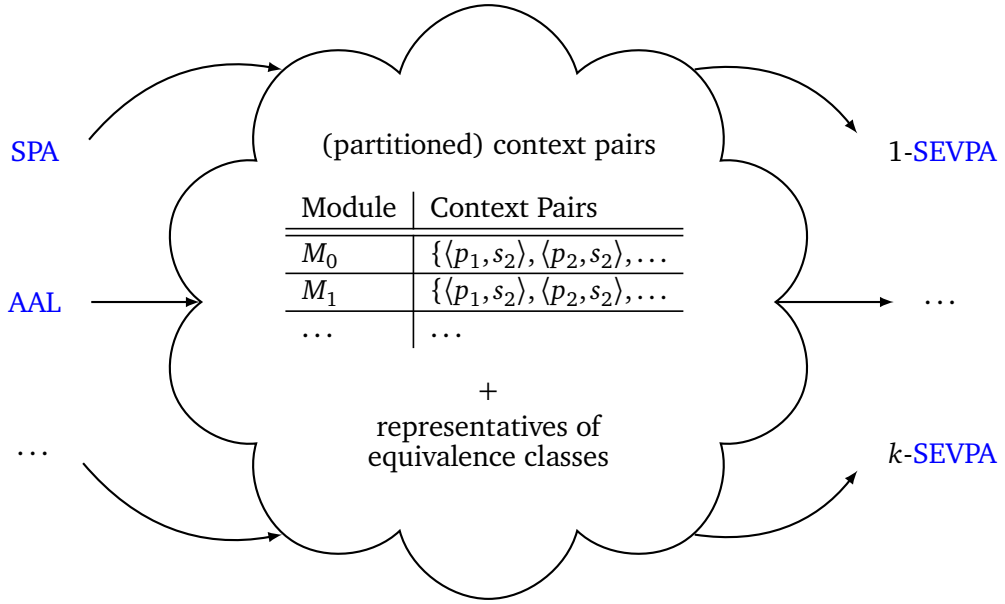


Figure 7.2
A SEVPA accepting the language $L = \{ccarcbrr\}$. Sink locations and corresponding transitions of the individual modules are omitted for readability.

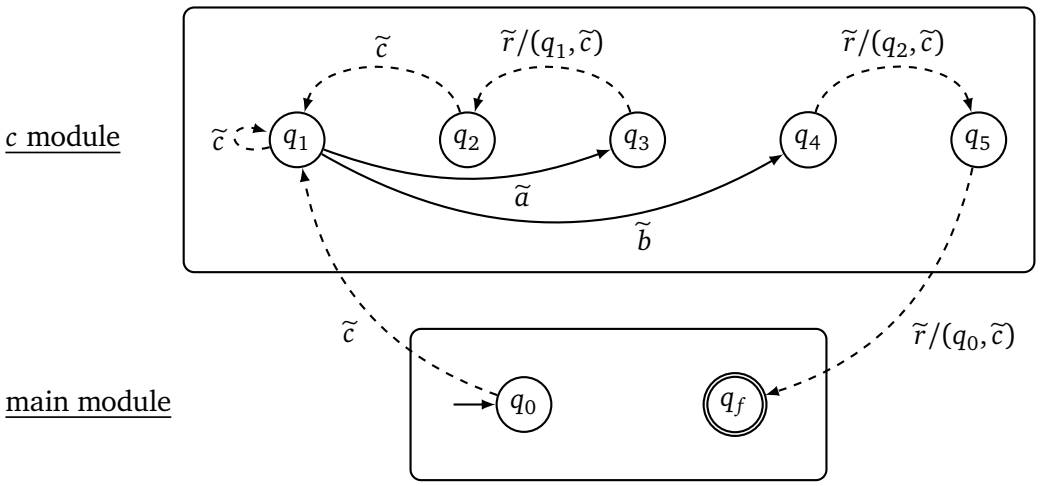
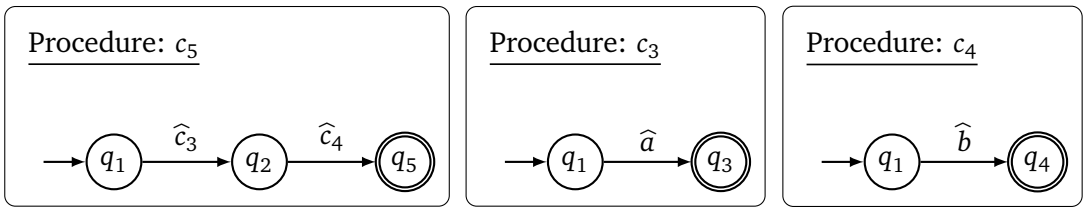


Figure 7.3
An SPA accepting the language $L = \{c_5c_3arc_4brr\}$. The initial procedure is c_5. Sink states and corresponding transitions of the individual procedures are omitted for readability.



follow the classic copy-rule semantics. When calling a procedure, the actions that follow the matching return symbol are already pre-determined by the current context and the invoked procedure cannot affect them (cf. Definition 29).

For representing a well-matched VPL via an SPA, the main challenge is to move the decision about the return successor from the return of a procedure to the invocation of a procedure. Let us look at the slightly adjusted language $L' = \{c_5c_3arc_4brr\}$. Here, the individual procedural behaviors are *de-aliased* by encoding the returning locations in separate call symbols. As shown in Figure 7.3, it is easy to see how this modification allows for an SPA-based representation.

De-aliasing is one of the major concepts to incorporate the expressiveness of SEVPAs into the SPA formalism. However, the refinement of call symbols creates a semantic gap between the original SEVPA/VPL and its SPA-based representation because the two formalisms no longer operate over the same input alphabet. This second issue can be tackled with a concept similar to *automated alphabet abstraction refinement (AAAR)* [86]. Given a word of a VPL, one may use a series of tests to determine for each occurrence of an aliased (or "abstract") call symbol, its de-aliased (or "concrete") representative. By repeating this step for all call symbols of a word, words over the abstract visibly push-down alphabet can be transformed into words over the concrete SPA input alphabet and thus use the SPA formalism to answer, e.g., membership questions for abstract SEVPA words. As each concrete call symbol has a unique, abstract source symbol, this process is easily reversible in order to map concrete words (generated by an SPA) to abstract words again.

Please note that, technically, the above concepts do not allow for *true* language equivalence between arbitrary SEVPAs and SPAs. Even well-matched VPLs can start with arbitrary call symbols (multiple different ones or none at all) and do not have to terminate with a return symbol. In contrast, SPA languages are necessarily minimally well-matched (cf. Definitions 29 and 32). This discrepancy can be easily compensated for by embedding the VPL of a SEVPA V in a distinct (main) call symbol and a return symbol, i.e., $\{main\} \cdot L(V) \cdot \{r\}$, and using this call symbol for the main procedure of the transformed SPA. Since this is an *external* modification to the language that does not affect $L(V)$ itself, it is a mere technical tweak to achieve (formal) language equivalence. This modification does not require any internal constraints on the SEVPA/VPL structure itself.

We continue with the construction of de-aliased SPAs from given SEVPAs and then look at how alphabet abstraction refinement can be implemented via characterizing sets.

7.3.1 De-Aliasing

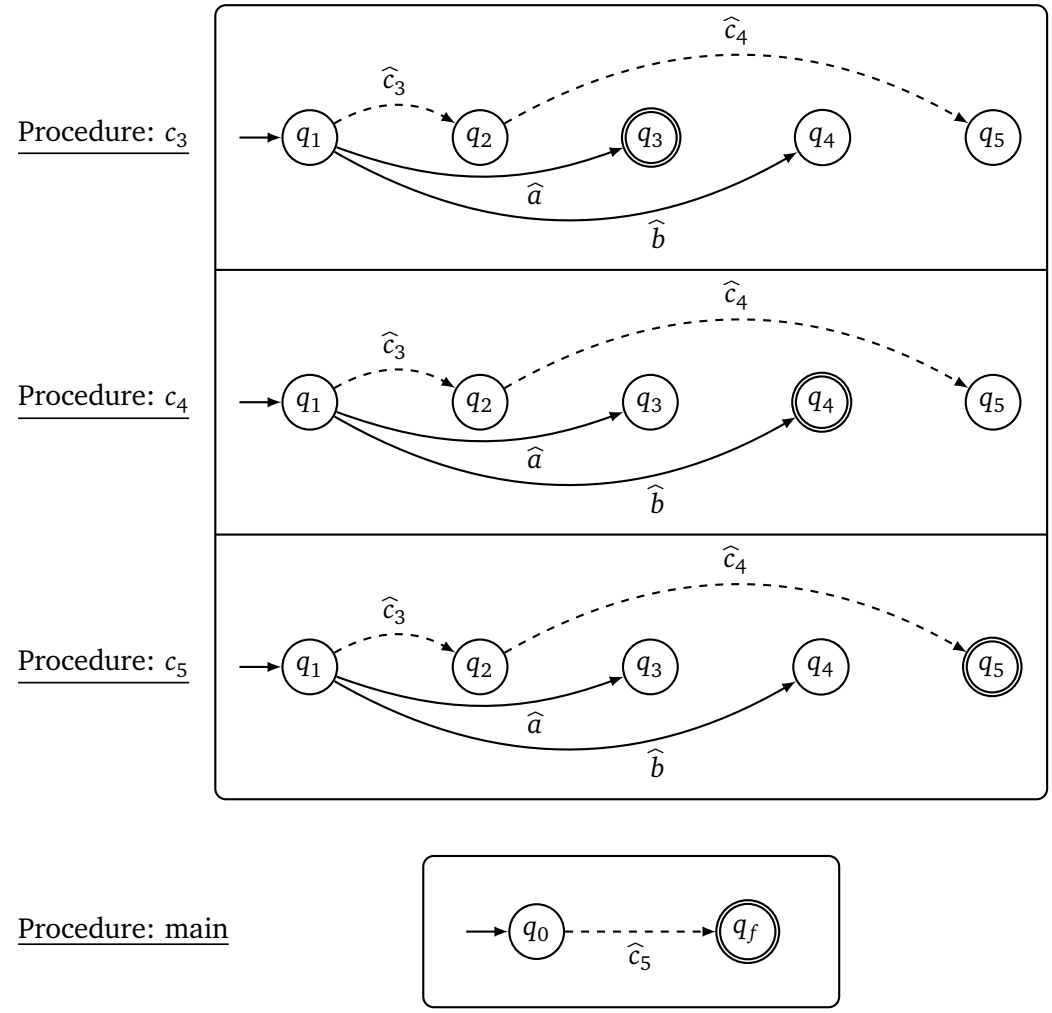
The following discussion assumes, for the ease of presentation, that the visibly push-down alphabets of the concerned SEVPAs only contain a single return symbol, i.e., only SEVPAs over SPA input alphabets are considered. It should be noted that the presented concepts can be easily generalized to arbitrary visibly push-down alphabets by considering a separate concrete call symbol for each possible combination of a location and a return symbol. Furthermore, only SEVPAs in the form of n-SEVPAs (where n denotes the number of call symbols) are considered, i.e., there exists a separate module for each individual (abstract) call symbol. This allows one to identify a module by its respective call symbol as opposed to a partition class index. Note that this is only a technical detail, as each well-matched VPL has an equivalent k-SEVPA representation for all $k \in \{1, \ldots, n\}$ [12].

Recall that in all runs of a k-SEVPA, each call from a module M_x to a module M_y must have its matching return symbol transition the k-SEVPA from module M_y back to module M_x again (cf. Definition 72). This means, when calling a procedure $\widetilde{c} \in \widetilde{\Sigma}_{call}$ in a location $q \in M_x$ (which fixes the top-of-stack symbol to $\langle q, \widetilde{c} \rangle$), there only exist a maximum of $|M_y|$ distinct return successors (if each $q' \in M_y$ selects a different return successor in M_x). Therefore, when trying to determine the return successor at the call of module M_y for the SPA-based semantics, it is only necessary to distinguish between $|M_y|$ different behaviors. This directly leads to the idea to introduce a separate concrete call symbol (to M_y) for each location of the module M_y. Each of the corresponding procedures for these concrete call symbols then represent a run of M_y that terminates at the respective location. As a result, the procedures are structurally similar to the module M_y, except for the accepting states which are determined based on the source of the encoded return transitions. To give an example of this transformation, Figure 7.4 shows this concept applied to the SEVPA of Figure 7.2.

For each location $q_i \in M_c$, there exist distinct procedures that exhibit similar structural properties to the module c. They each have the same amount of states (locations) and an identical transition relation for internal alphabet symbols. Return transitions are omitted because they are implicitly encoded by the acceptance of states. For call transitions, they incorporate the proposed mechanism to explicitly decide return successors at the call symbol. For example, take the return transition $\langle q_3, \widetilde{r}, \langle q_1, \widetilde{c} \rangle, q_2 \rangle \in \delta_{ret}$ of the SEVPA-based representation in Figure 7.2. It states that if one has called module c from q_1 (top-of-stack symbol) and parses r while being in q_3, the SEVPA should transition into q_2. In the SPA-based representation, this equates to the (intra-procedure) transition from q_1 to q_2 when calling the refinement of module c that terminates in q_3 (c_3). By choosing the concrete call symbol that corresponds to the source location of the respective return transition, one ensures to only model valid return transitions of the SEVPA-based representation. With similar reasoning, the corresponding c_4 and c_5 transitions are chosen.

The main module (main procedure) constitutes a special case. By definition [12], the main module M_0 of a k-SEVPA does not contain any meaningful return transitions. Consequently, the previous transformations are not necessary for constructing the main procedure. Instead, it suffices to use the acceptance of locations of M_0 to identify the

Figure 7.4
The SEVPA of Figure 7.2 transformed into an SPA. Sink states, unreachable procedures, and corresponding transitions are omitted for readability.



accepting states of the main procedure, as the acceptance of a word in the context of a SEVPA directly corresponds to successfully returning from the main procedure in the context of an SPA. Note that explicitly representing the main module as a distinct procedure also requires one to introduce an additional initial call symbol for the SPA-based representation. Here, the previously mentioned "*main*" call symbol is used for embedding the concerned VPLs in order to make it minimally well-matched.

In the following, we continue with a formalization of the transformation sketched above. We first look at the transformation from abstract alphabet symbols to concrete alphabet symbols in order to initialize the de-aliasing process for a given SEVPA. Afterwards, we look at the transformation of individual modules to multiple procedures and aggregate these procedures to a complete SPA to finish the transformation.

Definition 77 (Alphabet concretization)
Let Σ be an SPA input alphabet with n call symbols and let "main" denote a designated call symbol for embedding a VPL. Let V be an n-SEVPA over Σ with modules $M_0, \ldots, M_n$. We define the concretized SPA input alphabet *as a tuple $\breve{\Sigma} = \breve{\Sigma}_{call} \uplus \Sigma_{int} \uplus \{r\}$ (with respect to V) such that*

$$\breve{\Sigma}_{call} = \{c_{i,j} \mid c_i \in \Sigma_{call}, q_j \in M_i, i \in \{1, \ldots n\}\} \cup \{main\}.$$

We write $\breve{\Sigma}_{proc} = \breve{\Sigma}_{call} \uplus \Sigma_{int}$ to denote the procedural (sub-) alphabet *of $\breve{\Sigma}$.*

Similar to SPAs, we use a special markup token $\breve{}$ to denote when input symbols, words, or alphabets are interpreted in a concretized context. If the call symbol is clear from the context, e.g., when the symbol itself is unique like in Figure 7.4, the index i may be omitted for a simpler notation.

Definition 78 (Module concretization)
Let Σ be an SPA input alphabet with n call symbols and V be a (total) n-SEVPA over Σ with modules $M_0, \ldots, M_n$. Let $\breve{\Sigma}$ be the concretized SPA input alphabet (with respect to V) and let $M_i = \{q_0, \ldots, q_k\}$ be the set of locations of module $i \in \{1, \ldots, n\}$ where q_i^* denotes the designated entry-point of the module M_i. The module concretization $\breve{M}_i$ *of M_i induces a set of procedural automata $\{\breve{P}^{c_{i,j}} \mid q_j \in M_i\}$ such that for $\breve{P}^{c_{i,j}} = \langle \breve{Q}^{c_{i,j}}, \breve{q}_0^{c_{i,j}}, \widehat{\breve{\Sigma}}_{proc}, \breve{Q}_F^{c_{i,j}}, \breve{\delta}^{c_{i,j}} \rangle$ we have*

- $\breve{Q}^{c_{i,j}} = M_i$,
- $\breve{q}_0^{c_{i,j}} = q_i^*$,
- $\breve{\delta}^{c_{i,j}} = \breve{\delta}_{call}^{c_{i,j}} \uplus \breve{\delta}_{int}^{c_{i,j}}$ *with*
  - $\breve{\delta}_{call}^{c_{i,j}} = \{\langle q_u, \widehat{c}_{x,y}, q_v \rangle \in \breve{Q}^{c_{i,j}} \times \widehat{\breve{\Sigma}}_{call} \times \breve{Q}^{c_{i,j}} \mid \langle q_y, \widetilde{r}, \langle q_u, \widetilde{c}_x \rangle, q_v \rangle \in \delta_{ret}\}$,
  - $\breve{\delta}_{int}^{c_{i,j}} = \delta_{int}$,
- $\breve{Q}_F^{c_{i,j}} = \{q_j\}$.

Since we consider deterministic SEVPAs, there only exists a single q_v for all possible pairs of $q_y, \widetilde{r}$ and $\langle q_u, \widetilde{c}_x \rangle$. Hence, the transition functions of the induced procedural automata are deterministic as well. However, the induced procedural automata are not total as transitions for the call symbol "main" are undefined.

Definition 79 (Main-module concretization)
Let Σ be an SPA input alphabet with n call symbols and V be a (total) n-SEVPA over Σ with modules $M_0, \ldots, M_n$. Let $\breve{\Sigma}$ be the concretized SPA input alphabet (with respect to V). The main-module concretization *$\breve{M}_0$ of M_0 induces a procedural automaton $\breve{P}^{main}$ such that for $\breve{P}^{main} = \langle \breve{Q}^{main}, \breve{q}_0^{main}, \widehat{\breve{\Sigma}}_{proc}, \breve{\delta}^{main}, \breve{Q}_F^{main} \rangle$ we have*

- $\breve{Q}^{main} = M_0$,
- $\breve{q}_0^{main} = q_0 \in Q_{in}$,
- $\breve{\delta}^{main} = \breve{\delta}_{call}^{main} \uplus \breve{\delta}_{int}^{main}$ *with*
  - $\breve{\delta}_{call}^{main} = \{\langle q_u, \widehat{c}_{x,y}, q_v \rangle \in \breve{Q}^{main} \times \widehat{\breve{\Sigma}}_{call} \times \breve{Q}^{main} \mid \langle q_y, \widetilde{r}, \langle q_u, \widetilde{c}_x \rangle, q_v \rangle \in \delta_{ret}\}$,
  - $\breve{\delta}_{int}^{main} = \delta_{int}$,
- $\breve{Q}_F^{main} = Q_F$.

The same remarks about determinism and totality of Definition 78 hold for the main-module concretization.

Definition 80 (SEVPA concretization)
Let Σ be an SPA input alphabet with n call symbols and V be a (total) n-SEVPA over Σ with modules $M_0, \ldots, M_n$. The concretization *of V over $\breve{\Sigma}$ is defined by the tuple $\breve{S} = \langle \breve{P}_1, \ldots \rangle$ such that $\breve{P}_i \in \{\breve{M}_0\} \cup \left(\bigcup_{k \in \{1,\ldots,n\}} \breve{M}_k\right)$. The initial procedure of $\breve{S}$ is main $\in \breve{\Sigma}_{call}$.*

Note that, in general, the proposed concretization does not yield minimal SPAs. On the one hand, this concerns individual procedures. For example, consider the transformed SPA in Figure 7.4. Each (non-main) procedural automaton contains equivalent states as one easily sees when comparing the procedures with their minimized versions in Figure 7.3. One the other hand, this concerns the aggregation of procedures. For example, the concretization of the SEVPA of Figure 7.2 would also create procedures $\breve{P}^{c_1}$ and $\breve{P}^{c_2}$. In Figure 7.4, these procedures are omitted because no accepted word of the SPA contains the call symbols c_1 or c_2.

The lack of minimality is mainly due to the fact that VPLs are characterized via paths in VPAs and in the case of k-SEVPAs the decision whether a well-matched word belongs to a VPL is decided in the main module. The proposed concretization transforms modules individually based on their structural characteristics. Therefore, the transformation also includes paths that reach a rejecting state in the main module. However, in the (generative) language definition of SPAs (cf. Definition 29), the decision about admissible (or "accepting") runs is decided at the call symbols. This results in a filtering of such "sink-paths" which contain call symbols that cannot terminate in accepting locations.

This technical detail can be easily addressed by adding a post-processing step that first strips the (concretized) SPA input alphabet of unreachable call symbols and then totalizes and minimizes the remaining procedural automata with respect to the remaining SPA input alphabet. Applying this post-processing step would transform the SPA of Figure 7.4 into the SPA of Figure 7.3. However, specifically the discussion about the concretization equivalence in Section 7.3.3 continues to use the non-minimized representation produced by the proposed transformation because it allows for a more direct comparison between paths in a SEVPA and paths in its concretized SPA version.

7.3.2 Alphabet Abstraction Refinement

The proposed transformation in Section 7.3.1 transforms a given SEVPA V over an SPA input alphabet Σ into a concretized SPA $\breve{S}$ over $\breve{\Sigma}$. As a result, a well-matched VPL $L(V)$ cannot be directly described by $\breve{S}$. This section presents a translation process that transforms words of a SEVPA V over Σ to words of $\breve{S}$ over the corresponding $\breve{\Sigma}$.

The main challenge of this process is determining for each abstract call symbol occurring in a word $w \in \Sigma^*$, which concrete call symbol it should be represented by in the concretized version. Summarizing the proposed alphabet concretization of Definition 77, each abstract call symbol $c_i \in \Sigma_{call}$ introduces concrete call symbols $c_{i,j} \in \breve{\Sigma}_{call}$ for each location $q_j \in M_i$. Consequently, the index for identifying the module (i) can already be inferred from the given c_i. What remains to be determined is the location q_j in which the run exits the module. This is a basic state (or location) identification problem: Given the well-matched (sub-) word starting with the abstract call symbol c_i and ending with its matching return symbol, which location is reached after parsing the (sub-) word?

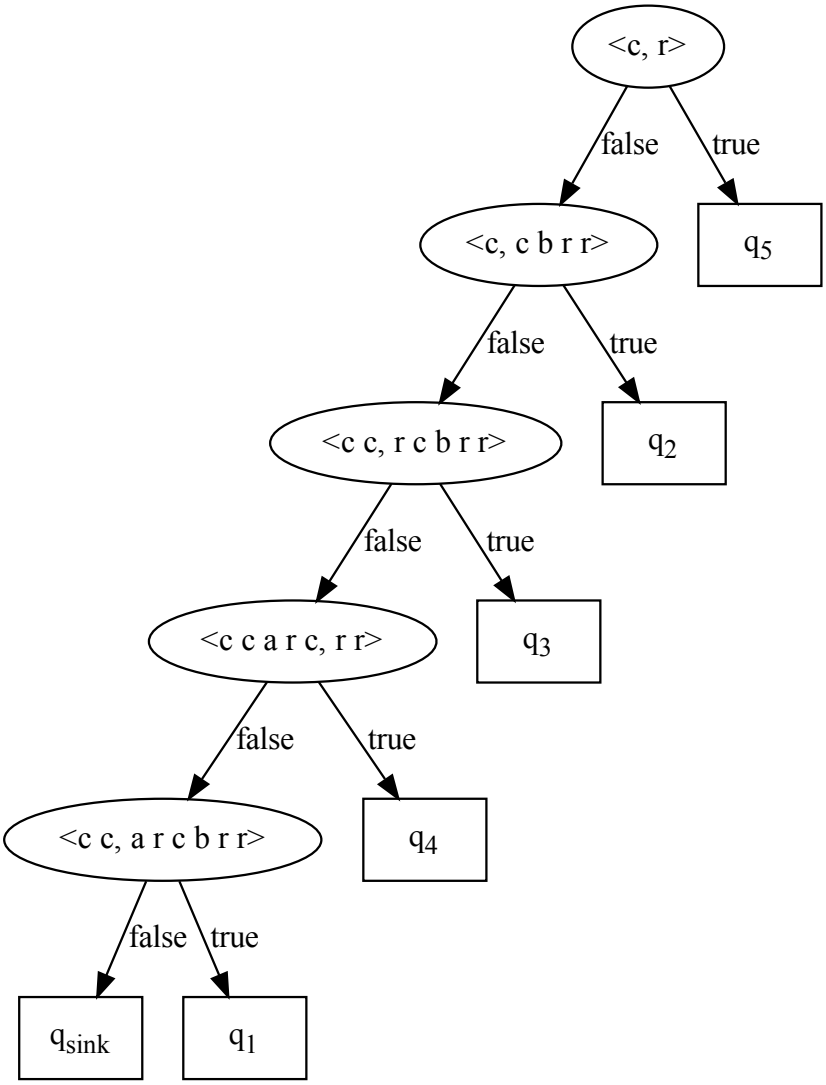
Since the SEVPA is available as a white-box model, it is possible to use existing techniques from the field of MBT to answer this question. The state (location) identification problem can be solved using characterizing sets (cf. Definition 23). Note that compared to the regular case, characterizing sets for SEVPAs consist of tuples of input words, as different locations can only be distinguished by using the syntactical congruence (cf. Definition 74). To give an intuition, Figure 7.5 presents the characterizing set of the module c of the SEVPA of Figure 7.2. For this example, the characterizing set is displayed as binary decision tree. Inner nodes represent elements of the characterizing set, whereas leaves represent locations of the SEVPA.

In order to determine the location that is reached by an input sequence $v \in WM(\Sigma)$, v is *sifted* through the decision tree such that for each inner node $\langle p, s \rangle$, the membership of $p \cdot v \cdot s$ in the language of V is tested. Depending on the answer, the sifting process continues with either the "true" successor or the "false" successor until a leaf node is encountered which identifies the location that v reaches.

Remark 6
This thesis does not go into details about computing characterizing sets of SEVPAs as this topic is beyond its scope. However, note that AAL and MBT share a lot of similar concepts. One can exploit the fact that both fields need to deal with the problem of state identification. While MBT has white-box access to automaton models, AAL needs external counterexamples in order to separate states.
For example, the characterizing set of Figure 7.5 has been extracted from the discrimination tree of the VPA adaption of the TTT algorithm [96] presented in [93, Chapter 6]. Since the SEVPA is available as a white-box model, it is possible to provide the AAL algorithm with a white-box equivalence query oracle (EQO) to guarantee a correct and complete set of discriminators.

In order to use characterizing sets for the translation of full words w of a well-matched VPL, w is processed in a symbol-wise fashion such that the state (location) identification problem is solved for every occurrence of a call symbol. For the exemplary word $ccarcbrr$

Figure 7.5
The characterizing set for locations of module c of the SEVPA of Figure 7.2. The checking sequences are organized in a tree-like fashion, essentially representing a decision tree for determining locations. q_{sink} denotes the sink location that has been omitted from Figure 7.2.



of Figure 7.2 and the characterizing set of Figure 7.5, this process is as follows:

- The first call symbol c has its matching return symbol r at the very end of the input word. Therefore, the inner sub-sequence consists of $carcbr$. Sifting this sub-sequence through the decision tree results in reaching the leaf q_5. As a result, the first call symbol c is translated to c_5.
- The second call symbol c has its matching return symbol prior to the third call symbol. Therefore, the inner sub-sequence consists of a. Sifting this sub-sequence through the decision tree results in reaching the leaf q_3. As a result, the second call symbol c is translated to c_3.
- The third and final call symbol c has ist matching return symbol at the last but one position of the input word. Therefore, the inner sub-sequence consists of b. Sifting this sub-sequence through the decision tree results in reaching the leaf q_4. As a result, the third call symbol c is translated to c_4.

Using the above technique, the abstract SEVPA-based input word $ccarcbrr$ is transformed to the concrete SPA-based input word $c_5c_3ac_4brr$. When embedding the abstract word in the designated *main* symbol and return symbol, it is easy to see how both words behave equivalently in their respective formalisms (cf. Figures 7.2 and 7.4). We continue with the formalization of this translation process.

Definition 81 (Location identification function)
Let Σ be an SPA input alphabet with n call symbols and V be a (total) n-SEVPA over Σ. Let M_i be a module of V for $i \in \{0,\dots,n\}$ where q_i^* denotes the designated entry-point of the module. Let $CS(M_i)$ be a characterizing set of module M_i and let $w = a_1 \cdot \ldots \cdot a_k \in WM(\Sigma)$ be a well-matched word. We define the location identification function *$LI_{M_i} : WM(\Sigma) \to \{c_{i,\cdot} \in \breve{\Sigma}_{call}\}$ such that $LI_{M_i}(w) = c_{i,j}$ iff w induces a run $\tau = \langle q_i^*, \sigma_0\rangle, \dots, \langle q_k, \sigma_k\rangle$ for some $\sigma_i \in ST(\Gamma_{VPA})$ and $CS(M_i)$ classifies q_k as q_j. This means LI_{M_i} identifies the location that is reached by w when parsed from the designated entry point of module M_i.*

As the previous example shows, the location identification function may be implemented on the basis of sifting words through a (characterizing) decision tree. The translation process is then defined by a symbol-wise processing of abstract well-matched VPL words, which replaces each (abstract) call symbol with its concretized representative.

Definition 82 (Abstract translation)
Let Σ be an SPA input alphabet and $w = u \cdot v$ be an abstract well-matched word over Σ with $w \in \Sigma^+, u \in \Sigma, v \in \Sigma^*$. The abstraction translation *is a function $\kappa : WM(\Sigma) \to WM(\breve{\Sigma})$ such that*

$$\kappa(\varepsilon) = \varepsilon,$$
$$\kappa(u \cdot v) = \begin{cases} LI_{M_i}(v[,\rho_v(1)]) \cdot \kappa(v) & \text{if } u = c_i \in \Sigma_{call} \\ u \cdot \kappa(v) & \text{otherwise.} \end{cases}$$

It is easy to see how the proposed translation transforms symbols within their classes, i.e., call symbols to call symbols, et cetera. As a result, the resulting concretized words are well-matched as well.

It is worth noting that when evaluating the location identification function of the abstract translation, one needs to query the original SEVPA for performing the membership tests. Recall from Section 7.3 that one of the key contributors to the expressiveness of SEVPAs is the fact that return successors can be determined *after* a run throughout a module whereas SPAs need to determine the return successor *upon* calling a procedure. It is not possible to encode this information directly in an SPA model. While the concretized SPA is completely detached from the original SEVPA, translating abstract words to concrete words keeps a (hidden) dependency on the original SEVPA. This is due to the way the alphabet concretization is implemented. A similar effect occurs with the original concept of AAAR [86] as well.

Furthermore, recall from Section 7.3.1 the fact that concretized SPAs are, in general, not minimal. When minimizing a concretized SPA like in Figure 7.3, one can also simplify the respective characterizing sets of the modules by removing elements that characterize obsolete procedures. For example, in the case of Figure 7.3 it is not necessary to identify c_1 and c_2 since neither call symbols occur in any accepted word of the concretized SPA. Instead, they may be aggregated to a single c_{sink} symbol that is not part of the final (concretized) SPA input alphabet so that words containing this call symbol are rejected automatically by $\breve{S}$. This modification also allows one to reduce the number of tests that the respective LI_{M_i} functions have to perform since elements such as $\langle c, c \cdot b \cdot r \cdot r \rangle$ or $\langle c \cdot c, a \cdot r \cdot c \cdot b \cdot r \cdot r \rangle$ can be removed from the characterizing set (or decision tree, cf. Figure 7.5).

7.3.3 Concretization Equivalence

This (sub-) section concludes the transformation of SEVPAs into SPAs by showing that the combination of the proposed concretization and the abstract translation allows one to represent an arbitrary well-matched VPL (embeded in the context of a *main* procedure) via an SPA-based formalism. First, we look at some utility lemmas before continuing with the main theorem of this section.

Lemma 8
Let Σ be an SPA input alphabet and $w = w_1 \cdot w_2$ with $w_1, w_2 \in WM(\Sigma)$. We have

$$\begin{aligned} \alpha(w_1 \cdot w_2) &= \alpha(w_1) \cdot \alpha(w_2), \\ \kappa(w_1 \cdot w_2) &= \kappa(w_1) \cdot \kappa(w_2), \text{ and} \\ \alpha(\kappa(w_1 \cdot w_2)) &= \alpha(\kappa(w_1)) \cdot \alpha(\kappa(w_2)). \end{aligned}$$

Proof. The first two equalities directly follow from Definitions 37 and 82 and the fact that w_1 and w_2 are independently well-matched. The third equality is a direct consequence of the nested application of the previous two equalities. □

Lemma 9
Let Σ be an SPA input alphabet with n call symbols and V be a (total) n-SEVPA over Σ with modules $M_0, \ldots, M_n$. Let $\breve{\Sigma}$ be the concretized SPA input alphabet (with respect to V)

and $\breve{S}$ be the concretized (non-minimized) SPA over $\breve{\Sigma}$. Let $w = a_1 \cdot \ldots \cdot a_k \in WM(\Sigma)$ and $\langle q_j, \sigma_0 \rangle, \ldots, \langle q_{j'}, \sigma_k \rangle$ be a w-induced run of module M_i for $i \in \{0, \ldots, n\}$ with $q_j, q_{j'} \in M_i$ and $\sigma_l \in ST(\Gamma_{VPA})$ for $l \in \{0, \ldots, k\}$, i.e, w induces a run starting in location q_j and terminating in location $q_{j'}$. Then we have

$$\breve{\delta}^{c_{i,\cdot}}(q_j, \alpha(\kappa(w))) = q_{j'}$$

or

$$\breve{\delta}^{main}(q_j, \alpha(\kappa(w))) = q_{j'}$$

respectively, i.e., the concretized, projected, w-induced run connects the same states (identified via their respective locations) in all M_i-induced concretizations $\breve{P}^{c_{i,\cdot}}$ ($\breve{P}^{main}$) as well.

Proof. We prove the cases for M_i with $i \in \{1, \ldots, n\}$ via structural induction over w. By using $\breve{P}^{main}$ instead, the case for the main module M_0 is shown analogously.

- Let $w = a_1 \cdot \ldots \cdot a_k \in \Sigma^*_{int}$. By Definition 82, we have $\kappa(w) = w = a_1 \cdot \ldots \cdot a_k \in \Sigma^*_{int}$ and by Definition 37 we have $\alpha(w) = \widehat{a}_1 \cdot \ldots \cdot \widehat{a}_k \in \widehat{\Sigma}^*_{int}$. By Definition 78, all internal transitions of $\breve{P}^{c_{i,\cdot}}$ are identical to M_i, so that $\widehat{w}$ traverses the same states (locations) as M_i, concluding the statement.
- Let $w = c_x \cdot v \cdot r$ for some $c_x \in \Sigma_{call}, v \in WM(\Sigma)$ such that the induction hypothesis holds for v. We know that the w-induced run transitions V from q_j to the initial location of module M_{c_x} to some location $q_y \in M_{c_x}$ and via a return transition $\langle q_y, \widetilde{r}, \langle q_j, \widetilde{c}_x \rangle, q_{j'} \rangle \in \delta_{ret}$ to $q_{j'}$. By Definition 82, we have $\kappa(w) = \breve{w} = c_{x,y} \cdot \kappa(v) \cdot r$ and by Definition 37 we have $\alpha(\breve{w}) = \widehat{c}_{x,y}$. By Definition 78, the previous return transition results in a $\widehat{c}_{x,y}$-labeled call transition from the state q_j to $q_{j'}$ in all $\breve{P}^{c_{i,\cdot}}$ and therefore
$$\breve{\delta}^{c_{i,\cdot}}(q_j, \alpha(\kappa(w))) = \breve{\delta}^{c_{i,\cdot}}(q_j, \widehat{c}_{x,y}) = q_{j'}.$$
- Let $w = w_1 \cdot w_2$ with $w_1, w_2 \in WM(\Sigma)$ such that the induction hypothesis holds for w_1, w_2. Since both w_1 and w_2 are independently well-matched, this means that the w-induced run of M_i consists of $\langle q_j, \sigma_0 \rangle, \ldots, \langle q_{j''}, \sigma_m \rangle, \ldots, \langle q_{j'}, \sigma_k \rangle$ with $q_{j''} \in M_i$ and $m = |w_1|$. By induction hypothesis, we know that
$$\breve{\delta}^{c_{i,\cdot}}(q_j, \alpha(\kappa(w_1))) = q_{j''} \text{ and } \breve{\delta}^{c_{i,\cdot}}(q_{j''}, \alpha(\kappa(w_2))) = q_{j'}.$$
Since the generalized transition function of DFAs is defined via repeated nested applications (cf. Definition 8), we can conclude using Lemma 8 that
$$\breve{\delta}^{c_{i,\cdot}}(q_j, \alpha(\kappa(w))) = \breve{\delta}^{c_{i,\cdot}}(q_j, \alpha(\kappa(w_1)) \cdot \alpha(\kappa(w_2))) = \breve{\delta}^{c_{i,\cdot}}(q_{j''}, \alpha(\kappa(w_2))) = q_{j'}.$$

□

Lemma 10

Let Σ be an SPA input alphabet with n call symbols and V be a (total) n-SEVPA over Σ with modules $M_0, \ldots, M_n$. Let $\breve{\Sigma}$ be the concretized SPA input alphabet (with respect to V) and $\breve{S}$ be the concretized (non-minimized) SPA over $\breve{\Sigma}$. Let $w = a_1 \cdot \ldots \cdot a_k \in WM(\Sigma)$ and $\langle q_j, \sigma_0 \rangle, \ldots, \langle q_{j'}, \sigma_k \rangle$ be a w-induced run of module M_i for $i \in \{0, \ldots, n\}$ with $q_j, q_{j'} \in M_i$ and

$\sigma_l \in ST(\Gamma_{VPA})$ for $l \in \{0,\dots,k\}$, i.e, w induces a run starting in location q_j and terminating in location $q_{j'}$. Then we have

$$(\alpha(\kappa(w)) \cdot \widehat{r}, \sigma)_{SPA} \xrightarrow{\kappa(w)}{}^{*} (\widehat{r}, \sigma)_{SPA}$$

for some $\sigma \in ST(\Gamma_{SPA})$, i.e., given the configuration $(\alpha(\kappa(w)) \cdot \widehat{r}, \sigma)_{SPA}$, the language-SOS of $\breve{S}$ can emit $\kappa(w)$.

Proof. We prove the cases for M_i with $i \in \{1,\dots,n\}$ via structural induction over w. By using $\breve{P}^{main}$ instead, the case for the main module M_0 is shown analogously.

- Let $w = a_1 \cdot \ldots \cdot a_k \in \Sigma^*_{int}$. By Definition 82, we have $\kappa(w) = w = a_1 \cdot \ldots \cdot a_k \in \Sigma^*_{int}$ and by Definition 37 we have $\alpha(w) = \widehat{a}_1 \cdot \ldots \cdot \widehat{a}_k \in \widehat{\Sigma}^*_{int}$. The statement then directly follows by repeated applications of int-rules.
- Let $w = c_x \cdot v \cdot r$ for some $c_x \in \Sigma_{call}, v \in WM(\Sigma)$ such that the induction hypothesis holds for v. We know that the w-induced run transitions V from q_j to the initial location of module M_{c_x} to some location $q_y \in M_{c_x}$ and via a return transition $\langle q_y, \widetilde{r}, \langle q_j, \widetilde{c}_x\rangle, q_{j'}\rangle \in \delta_{ret}$ to $q_{j'}$. By Definition 82, we have $\kappa(w) = \breve{w} = c_{x,y} \cdot \kappa(v) \cdot r$ and by Definition 37 we have $\alpha(\breve{w}) = \widehat{c}_{x,y}$. For the v-induced (sub-) run, we know by Lemma 9 that $\breve{\delta}^{c_{x,y}}(q^*_x, \alpha(\kappa(v))) = q_y$ and by construction of $\breve{P}^{c_{x,y}}$ we know that $q_y \in \breve{Q}_F^{c_{x,y}}$, hence $\alpha(\kappa(v)) \in L(\breve{P}^{c_{x,y}})$. By application of a call-rule and the induction hypothesis we can then show

  $$(\widehat{c}_{x,y} \cdot \widehat{r}, \sigma)_{SPA} \xrightarrow{c_{x,y}} (\alpha(\kappa(v)) \cdot \widehat{r}, \widehat{r} \bullet \sigma)_{SPA} \xrightarrow{\kappa(v)}{}^{*} (\widehat{r}, \widehat{r} \bullet \sigma)_{SPA} \xrightarrow{r} (\widehat{r}, \sigma)_{SPA}$$

  which concludes the statement.
- Let $w = w_1 \cdot w_2$ with $w_1, w_2 \in WM(\Sigma)$ such that the induction hypothesis holds for w_1, w_2. Since both w_1 and w_2 are independently well-matched, this means that the w-induced run of M_i consists of $\langle q_j, \sigma_0\rangle, \dots, \langle q_{j''}, \sigma_m\rangle, \dots, \langle q_{j'}, \sigma_k\rangle$ with $q_{j''} \in M_i$ and $m = |w_1|$. Given Lemma 8, we can write $\alpha(\kappa(w))$ as $\alpha(\kappa(w_1)) \cdot \alpha(\kappa(w_2))$. Given the configuration $(\alpha(\kappa(w_1)) \cdot \alpha(\kappa(w_2)) \cdot \widehat{r}, \sigma)_{SPA}$, the SOS-system processes the state of a configuration in a symbol-wise fashion so that we can conclude with the argumentation from the previous two cases that

  $$(\alpha(\kappa(w_1)) \cdot \alpha(\kappa(w_2)) \cdot \widehat{r}, \sigma)_{SPA} \xrightarrow{\kappa(w_1)}{}^{*} (\alpha(\kappa(w_2) \cdot \widehat{r}, \sigma)_{SPA} \xrightarrow{\kappa(w_2)}{}^{*} (\widehat{r}, \sigma)_{SPA}$$

  which concludes the statement. □

Theorem 16 (Concretization equivalence)
Let Σ be an SPA input alphabet with n call symbols and V be a (total) n-SEVPA over Σ with modules $M_0,\dots,M_n$. Let $\breve{\Sigma}$ be the concretized SPA input alphabet (with respect to V) and $\breve{S}$ be the concretized (non-minimized) SPA over $\breve{\Sigma}$. Then we have

$$w \in L(V) \Leftrightarrow main \cdot \kappa(w) \cdot r \in L(\breve{S})$$

for all $w \in WM(\Sigma)$.

Proof. "⇒": Since we have $w \in L(V)$, we know that there exists a w-induced run from the initial location q_0 of V to an accepting location q_f of V. By Lemma 9, we know that $\breve{\delta}^{main}(q_0, \alpha(\kappa(w))) = q_f$ and by Definition 79, we know that $q_f \in \breve{Q}_F^{main}$. Therefore, we have $\alpha(\kappa(w)) \in L(\breve{P}^{main})$. With Lemma 10 we can then conclude

$$(\widehat{main}, \bot)_{SPA} \xrightarrow{main} (\alpha(\kappa(w)) \cdot \widehat{r}, \varepsilon \bullet \bot)_{SPA} \xrightarrow{\kappa(w)}{}^{*} (\widehat{r}, \varepsilon \bullet \bot)_{SPA} \xrightarrow{r} (\varepsilon, \bot)_{SPA}$$

and by Definition 29 we have $main \cdot \kappa(w) \cdot r \in L(\breve{S})$.
"⇐": We show via contraposition that $w \notin L(V) \Rightarrow main \cdot \kappa(w) \cdot r \notin L(\breve{S})$. Since we have $w \notin L(V)$, we know that there exists a run from the initial location q_0 of V to a rejecting location q_r of V. By Lemma 9, we know that $\breve{\delta}^{main}(q_0, \alpha(\kappa(w))) = q_r$ and by Definition 79, we know that $q_r \notin \breve{Q}_F^{main}$. Since $\breve{P}^{main}$ is constructed from a deterministic n-SEVPA, there exists no other (accepting) state that $\alpha(\kappa(w))$ reaches. Therefore, we have $\alpha(\kappa(w)) \notin L(\breve{P}^{main})$. Hence, the word $main \cdot \kappa(w) \cdot r$ contains a rejected, projected procedural invocation and by negation of Theorem 1, we can conclude that $main \cdot \kappa(w) \cdot r \notin L(\breve{S})$. □

Essentially, Theorem 16 states that only for accepted, w-induced runs of a VPA V, there exist $main \cdot \kappa(w) \cdot r$ paths in the language-SOS system of $\breve{S}$ reaching the final configuration $(\varepsilon, \bot)_{SPA}$. As a result, the membership question for V can be answered on the basis of an SPA model using the proposed abstract translation function and the embedding into a designated main procedure.

Note that for the ease of argumentation, Lemmas 9 and 10 and Theorem 16 assume that the concretized SPA $\breve{S}$ is not minimized which allows for a direct correspondence between locations of V and states of $\breve{S}$. However, at the core of their proofs, the statements only require the language properties of procedural automata. If the procedural automata of $\breve{S}$ were canonical, e.g., due to a prior minimization, it would essentially only introduce an additional mapping of locations to states that one would need to account for. As a result, the proposed post-processing steps of Section 7.3.1 to cleanup the concretized SPA $\breve{S}$ does not affect its ability to describe (embedded) well-matched VPLs.

7.4 Discussions

This section discusses further ideas and concepts in relation to the transformations between SPAs and SEVPAs.

7.4.1 Return-Matched Visibly Push-Down Languages and Visibly Push-Down Transducers

As discussed in Section 7.1.2, the concept of SEVPAs provides canonical VPA models for well-matched VPLs, which makes them an interesting counterpart to SPAs for considering transformations. However, VPAs, in general, also allow for describing call-matched VPLs and return-matched VPLs. Especially for the latter class of languages, this thesis presents the concept of SBAs which represent (instrumented) context-free *behaviors* by means of

prefix-closed languages. The question arises whether a similar transformation can be established for return-matched VPLs. To the best knowledge of the author, there exists no work on learning or testing VPAs in the context of return-matched VPLs including possible consequences for the canonicity of VPA-based models. In particular, the work of Isberner [93] and his characterization via the unified congruence for well-matched VPLs (cf. Definitions 74 and 75) does not allow for return-matched languages. Since the constructed VPAs are essentially 0-SEVPAs (recall that the unified congruence merges the main module M_0 with the single call-module M_1 of a 1-SEVPA), one cannot separate well-matched words from return-matched words via the acceptance of locations alone. Finding relationships between SBAs and corresponding VPA-based formalisms is an interesting topic for future research but goes beyond the scope of this thesis.

With SPMMs, this thesis presents an interpretation of SBAs for modelling (instrumented) deterministic transductions that follow an incremental lock-step pattern, using a distinct Mealy-based representation. On the side of VPAs, a similar concept is implemented by so-called visibly push-down transducers (VPTs) [144]. VPTs extend VPAs by an output function that associates with each transition a possible output symbol. Via non-determinism, i.e., allowing ε-inputs and ε-outputs on transitions, VPTs allow for a powerful transduction formalism for which several closure properties known from regular VPAs / VPLs no longer hold. An extensive analysis of VPTs is given in [155] and further work by the author [56]. VPTs clearly supersede the expressiveness of SPMMs, specifically due to the support of non-determinism. Its impact is easy to see, as already for the regular case, non-deterministic Mealy machines are able to describe transductions that deterministic Mealy machines cannot. It is an interesting question whether the subclass of *deterministic VPTs with incremental lock-step output behavior* exhibit properties that allow for a comparison to or even a transformation into SPMMs. However, this question is also beyond the scope of this thesis and may be investigated separately in future research.

7.4.2 SPA-Based Learning of Visibly Push-Down Languages

With the ability to represent (embedded) well-matched VPLs via SPAs as presented in Section 7.3, the question arises whether the involved techniques can be directly integrated into the disciplines of MBQA. Specifically for AAL, the properties of SPA models with regard to, e.g., model size, may potentially improve the performance of the learning process. However, the task of de-aliasing abstract call symbols introduces an overhead that does not justify this approach.

The major issue of this approach is the fact that an SPA learner does not know the full concretized SPA input alphabet beforehand. Instead, it becomes a part of the inference process as well. During the exploration phase, this is not a problem as the procedural membership query oracles (MQOs) can easily map the concrete call symbols back to the abstract call symbols in order to evaluate expanded membership queries (MQs) on the VPL-based system under learning (SUL). However, during the verification phase, the learner receives a VPL-based counterexample that first needs to be concretized in order to refine the SPA hypothesis operating over the concretized SPA input alphabet. For this

task, the learner needs to account for the possibility to observe abstract call symbols for which the correct concretization has not yet been found. Based on the construction of the concretized SPA input alphabet (cf. Definition 77), the learner needs to keep track of the possible locations of the modules in order to determine the correct concrete call symbol.

At this point, the SPA learner needs to perform the same work of an n-SEVPA learner including additional work (queries) for the SPA inference. For example, consider for the SPA of Figure 7.4 the situation that c_3 gets split into c_3 and c_4 because the SPA learner has detected that there exist locations q_3, q_4 (cf. Figure 7.5). While the procedural learner of P^{c_4} can be started independently, all $\widehat{c}_3$ transitions in the hypotheses of the remaining learners need to be re-evaluated because the mapping of (the abstract) c has changed. Here, the alternative of using an n-SEVPA learner directly and using an offline transformation of the n-SEVPA into an SPA (which requires no additional queries) has the better query performance. However, a direct VPL-via-SPA approach may hold some benefits in scenarios where the full VPA semantics only extend to a few procedures or modules (cf. Section 11.2.2).

7.5 Summary

This section concludes the chapter by summarizing its main results.

- In general, VPAs are more expressive than SPAs because VPAs can describe (call-matched, return-matched, non-minimally well-matched) languages that SPAs cannot.
- SPA languages can be represented via VPAs by means of k-SEVPAs, which are a structurally constrained form of VPAs that support canonical representations and describe well-matched VPLs. The transformation is based on the canonical construction of k-SEVPAs using equivalences classes and representatives of the concerned language congruence(s) which can be inferred from a white-box analysis of the respective SPA.
- Embedded well-matched VPLs, i.e., minimally well-matched VPLs with a designated main procedure, can be represented by SPAs via a two-step transformation process:
  1. a concretization step of the respective VPAs de-aliases the behavior of call symbols, that depends on the reached locations within modules when returning from the call and
  2. a translation step of (abstract) VPL words to (concretized) SPA words allows one to query the transformed SPA formalisms for the original membership question.

CHAPTER 8

Related Work

This chapter presents related work from the field of model-based quality assurance (MBQA) concerning verification, testing, and learning of systems. As systems of procedural automata (SPAs), systems of behavioral automata (SBAs), and systems of procedural Mealy machines (SPMMs) represent model types, this chapter focuses on related work on procedural systems and possible relationships to the presented formalisms of this thesis. For a contrasting juxtaposition with the "competing" model type of visibly push-down automata (VPAs), see Chapter 7. For a discussion on the practical impact of the two competing model types (in the context of active automata learning (AAL)), see Chapter 10.

8.1 Model Verification

The field of model verification comprises a plethora of different formalisms and verification techniques [18, 47]. This is due to the fact that specifications involve a lot of different stakeholders which all have different requirements and expectations. Vardi [170] rightfully notes in his paper that specifications need to address several needs such as

- expressiveness, i.e., the possibility to describe the intended behaviors,
- usability or complexity, i.e., the ease of describing intended properties, and
- compositionality, i.e., the notion of closure when incrementally combining behavioral traits.

As a result, many popular specification formalisms can be found nowadays. Linear time-based logics such as the linear temporal logic (LTL) [140] interpret the global system behavior as a single (in-) finite linear sequence which progresses as the system runs. Branching-time logics such as the computational tree logic (CTL) [46] interpret a system by unrolling paths of the system in a tree-like fashion. There also exist combined approaches such as the computational tree logic with linear time assertions (CTL*) [52] which allows for an even more expressive formalism as neither LTL nor CTL subsumes the other. Some logics such as linear temporal logic with past (PLTL) [119] add new semantics in the form of past-modalities to linear-time specifications. See, e.g., [105] for a survey.

A common property of the previously mentioned logics is the fact that they interpret system behavior on the basis of state properties. As a result, the underlying model types for these logics are often based on Kripke transition systems (KTSs) or variations thereof. In contrast, logics such as the Hennessy-Milner logic (HML) [76] or action-based CTL

(ACTL) [132] introduce the notion of input modalities and express behavior via labeled transitions between system states. Here, the respective model types are often based on labeled structures such as labeled transition systems (LTSs) which focus on a different type of system semantics.

There also exist combined logics such as the (modal) μ-calculus [107] which supports specifying both state properties and input modalities and consequently operate on merged models types such as labeled KTSs. It is quite fittingly described by Burkart et al. [38] as "the assembly language for temporal logics" due to its low-level syntax and semantics, but its exceptional expressiveness. One can often find the (modal) μ-calculus at the core of model checker tools, as the μ-calculus subsumes logics such as LTL and CTL. By translating other logics to the (modal) μ-calculus, developers of model checkers can focus on a single implementation while offering more convenient specification "front-ends" to the end-users.

Due to the notion of rigorous (de-) composition of SPAs, SBAs, and SPMMs, the above logics and corresponding tools may be used for the verification of individual procedures. However, there has also been put a lot of effort into the global verification of context-free systems.

8.1.1 Context-Free Model Verification

Chapter 4 presents the transformations of SPAs, SBAs, and (via embedding in SBAs) SPMMs into context-free process systems (CFPSs) so that one can use the approach by Burkart et al. [37] to verify (alternation-free) modal μ-calculus formulae on SPA models and SBA models. Later work of the authors [38] considers generalizations of CFPSs that allow for model checking of the full modal μ-calculus.

Section 7.1 presents another transformation from SPAs into VPAs that opens up further possibilities for model verification. For example, the characteristics of VPAs can be equivalently described via recursive state machines (RSMs) [109]. Alur et al. present the CaRet [9] logic which allows one to verify linear-time properties on RSMs and consequently on words of visibly push-down languages (VPLs). CaRet integrates the special roles of call symbols and return symbols by offering special procedural modalities that allow one to explicitly address the beginning and the end of procedural invocations. This ability allows one to intuitively specify and verify inter-procedural and intra-procedural properties in a pre-condition and post-condition type of fashion. There exist several extensions to CaRet such as HyCaRet [33] which extends CaRet formulae by additionally allowing existential qualifiers, and visibly linear temporal logic (VLTL) [34] which generalizes[4] CaRet formulae to cover the full spectrum of VPLs. There also exist fixpoint-based verification logics for RSMs [8] and VPLs [32]. Furthermore, Alur et al. have presented a generalization of VPAs called nested word automata (NWAs) [10]. For NWAs there exist verification logics such as nested word temporal logic (NWTL) and NWTL$^+$ [13] which add support for new (procedural) modalities and operators. The execution semantics of RSMs can alternatively be expressed via push-down systems (PDSs) [7] as well.

[4] VLTL uses a different syntax than CaRet but CaRet formulae can be transformed into equivalent VLTL formulae in linear time [34].

For many of the procedural model types (CFPSs, PDSs, RSMs, VPAs), there exist a rich tool landscape [20, 73, 103, 161, 162, 163] that provides access to various verification algorithms that either use well-known logics or offer tool-specific formats as specification formalisms. With the ability to translate the semantics of (at least) SPAs to many of these formalisms, the above verification logics become applicable to the model types presented in this thesis as well.

8.2 Model-Based Testing

Model-based testing (MBT) [36, 111] and for this thesis specifically, conformance testing [68, 111], covers a variety of techniques which each target different goals and environments. Section 2.3 sketches the W-method [44] as an example of a conformance testing algorithm. This section puts this approach into context of other methods and discusses some adjustments depending on the properties at hand.

One major requirement of the W-method to be applicable is the possibility to reset a system so that a conformance test can consist of *multiple*, *independent* tests. In the context of modeling hardware systems or software systems, this is often a reasonable assumption, as systems usually can be rebooted or restarted. Furthermore, this property also aligns with the requirements of AAL, as AAL algorithms need to pose *multiple*, *independent* queries to the system, which is often implemented via a reset.

A straight-up improvement of the W-method is the partial W-method by Fujiwara et al. [65]. It splits the conformance test into different phases and introduces the concept of *state-local* characterizing sets which overall allow for the construction of fewer and shorter test cases. Both approaches can be extended to cover additional states that are not yet represented by a (hypothesis) model. If given an upper bound for the number of states of the implementation, both approaches can use this extension to construct conformance tests that answer the equivalence problem provably correct. This methodical and fine-grained analysis of models makes the two methods a powerful but also time-expensive approach for model-based testing (MBT). Specifically for AAL, where MBT may be used to implement equivalence query oracles (EQOs), Smetsers et al. [159] show that introducing fuzzing to the ((partial) W-method-based) construction of conformance tests can improve the performance of detecting in-equivalences.

Sometimes, hardware systems and software systems may have the possibility to emit status messages about the current state, e.g, when using a debug build of a software. In this case, the construction of a conformance test can be drastically simplified because there is no longer a need for an elaborate state characterization as this task can be directly implemented via status messages. Methods as simple as transition cover sets or transition tours [66] (if strongly-connected) mixed with status messages can then be used for conformance testing.

Sometimes, hardware systems and software systems may not be resettable or it may be expensive to do so. This step can be compensated for if the system exhibits other structural properties such as a strong connectedness, i.e., when all states are reachable from each other. Here, resets can be directly be replaced by *synchronizing sequences* [104, 138, 151]

which transition a system into a pre-determined, e.g., initial, state. Alternatively, *homing sequences* [80, 148, 151] may be used to transition the system into uniquely determinable states and continue with the characterization of these states. See, e.g., [68, Section 4.5], for an overview of alternatives depending on the systems characteristics. Note that while many methods for conformance testing are originally presented for systems with transition outputs such as Mealy machines, it is easy to adjust them to work with acceptor-based systems by, e.g., interpreting the acceptance of a successor state as the "output" of a transition.

With the notion of rigorous (de-) composition of SPAs, SBAs, and SPMMs, many of the concepts and extensions of MBT of regular systems can be lifted towards (instrumented) context-free systems. The concepts discussed in Chapter 5 provide a guidance for handling the specific semantics of the difference model types (such as transitions of non-continuable input symbol in case of SBAs and SPMMs) so that one can construct conformance tests for (instrumented) context-free systems that meet the individual requirements of the MBT scenario at hand.

8.3 Active Automata Learning

The seminal work on AAL was proposed by Angluin [15]. She introduced the concept of the minimally adequate teacher (MAT) framework which enables a learning algorithm to actively query a system (or an unknown language) for information by means of membership queries (MQs) and equivalence queries (EQs). With the LStar algorithm, she presented an algorithm for inferring regular formal languages (in polynomial time) based on this framework. Since then, the (MAT-based) field of AAL has experienced a plethora of improvements and extensions.

One dimension concerns the algorithmic aspects of learners. LStar uses an *observation table*, a table-based structure in which rows represent (not necessarily unique) access sequences to hypothesis states and columns represent distinguishing futures that separate the behavior of states. The cells of the table store answers of MQs that are constructed from the concatenation of the respective row-label and column-label. Consequently, this data-structure and hence the (query-) performance of the learner grows quadratically in the number of rows and columns. Kearns et al. [101] propose an AAL algorithm that manages a reduced set of in-equivalent access sequences of hypothesis states and uses a *discrimination tree* to organize the distinguishing futures. In practice, this reduces the multiplicative dependency between access sequences and distinguishing futures that LStar suffers from and allows the algorithm of Kearns and Vazirani to lower the number of queries for inferring a model.

Another aspect that affects the (query) complexity of learning algorithms is the handling of counterexamples. The original LStar algorithm [15] adds all prefixes of a counterexample to the rows of the observation table and uses the notion of closedness and consistency to construct hypotheses consistent with the observations. In a similar fashion, Maler et al. [116] propose to add all suffixes of a counterexample to the columns of the observation table whereas Shahbaz et al. [156] optimize this idea to only include

suffixes of the distinguishing future of a counterexample. Rivest et al. [146] propose a novel analysis of counterexamples that performs a binary search on transformations of the counterexample to extract only a single distinguishing future that can be added to the columns of the observation table. Together with an improved book-keeping of in-equivalent but *prefix-closed* access sequences, the proposed approach improves even the asymptotic query performance of the learning algorithm. An abstract framework for formalizing counterexample analysis and further analysis strategies are presented in [97].

Howar [84] combines the previous two major improvements with the proposal of the "observation pack" algorithm. It maintains a prefix-closed set of (in-equivalent) access sequences to enable the counterexample analysis of Rivest et al. [146]. The single suffixes extracted from counterexamples are then organized in a discrimination tree. The TTT algorithm by Isberner et al. [96] distills this concept by adding a post-processing step to ensure suffix-closure of distinguishing futures in order to reduce their redundancy, which further boosts the symbol performance of the learner in cases of long counterexamples. Recently, Howar et al. [85] proposed the concept of *lazy partition refinement* which incorporates the TTT effect on-the-fly. The impact of these changes can be seen in the comparisons with the original algorithms [85, 96].

Besides the algorithmic aspects of AAL, practical applicability plays another important role for the growing interest in this field of research. On the one hand, this is due to the growing expressiveness of models. Originally, Angluin [15] proposed her algorithm for the inference of regular languages, i.e., a deterministic finite acceptor (DFA)-based inference process being only able to describe binary properties. Hungar et al. [91] exploit the prefix-closure of systems to boost query performance and Margaria et al. [118] later generalize this concept to reactive systems by proposing an adaption of the LSTAR algorithm for inferring Mealy machines. Especially for describing hardware systems and software systems, data-management [4, 6, 30, 40, 51, 58, 86, 88, 94, 115] and recursion [93, 109] are essential for comprehensive descriptions of behavior. In situations where uncertainty or "noise" might be an issue, learning probabilistic automata [55, 164, 165] may be an adequate solution. Often, certain application domains allow AAL algorithms to exploit properties of the environment. For example, the "ADT" learner [64] for inferring Mealy machines exploits the direct output semantics of Mealy machines to incorporate adaptive distinguishing sequences [112] instead of fixed distinguishing suffixes to separate system states, improving the query performance of the learner in certain scenarios [122].

On the other hand, this is due to pragmatic solutions for bridging the gap between theoretical concepts and the practical problems. Concepts, such as filters [2, 6, 86, 117] for dealing with (data-) abstraction, caching [77, 117], or parallelism [77, 90] for boosting performance, lower the entry hurdle for applying AAL in real-world scenarios. Along with the development of various tools to support this process [6, 31, 39, 95, 102, 125, 127, 128, 169] there have been several success-stories of AAL in practical scenarios [1, 2, 6, 26, 41, 57, 91, 99, 131, 136, 143, 158, 167, 172].

As the learning of SPAs, SBAs, and SPMMs is not an isolated process, but rather a simultaneous process of multiple (regular) inference processes, learning (instrumented) context-free systems has a strong connection with the related work. Improvements in

the field of regular language inference also positively affect the inference process of instrumented context-free systems as shown in Chapter 10. Furthermore, the notion of rigorous (de-) composition may allow more of the regular extensions to be lifted to the context-free level, as discussed in Chapter 11.

8.3.1 Context-Free Active Automata Learning

Specifically for procedural systems, the work on the inference of context-free languages (CFLs) on the basis of queries already starts with Angluin's seminal work on AAL. However, the approach presented in [15] requires a special type of non-terminal MQ which, to the best knowledge of the author, has prevented any application in practical MBQA scenarios. Unfortunately, there exist rather negative results for the general inference of CFLs. [16] shows that there exists no polynomial-time algorithm for inferring CFLs with only EQs and [17] shows that there exists no polynomial-time algorithm for inferring CFLs with only MQs. The results of [114] conjecture that similar results may hold even if both types of queries are available. Positive results can be found if the expressiveness of languages is reduced to, e.g., simple deterministic languages (SDLs) [98]. However, note that [98] uses *extended EQs* which allow for more expressive hypothesis models (arbitrary CFLs) than ultimately inferred ones (SDLs).

Special attention should be given to whether the learning algorithms target CFLs or context-free grammars (CFGs). Especially CFL-focused learning algorithms may often expect systems to be described by a CFG in some kind of normal form, such as the Chomsky normal form [42] or the Greibach normal form [72]. While from a mere language perspective these transformations do not cause any problems, they do change the semantics of a system in the context of MBQA. Returning to Example 2, recall that there exist multiple different CFGs to describe palindromes over $\{a, b, c\}$ but it is a deliberate design decision to delegate the emission of cs to procedure G. These information get lost after transformations into certain normal forms.

A common theme for CFG-focused learning algorithms is that the interactions with the system under learning (SUL) are enriched by some form of structural information. For example, [150] uses *structural MQs* and *structural EQs* to infer tree automata for describing the *original* CFG of a system. Similar approaches can also be found in the field of *passive learning* of CFGs [67, 149]. These specialized queries (or training samples in case of passive learning) share a lot of concepts with the proposed instrumentation of this thesis, as they allow the learner to extract crucial information about the scope of a procedure or non-terminal, respectively. However, it may be argued that providing these information via the inherent observable language of the system, e.g., via call symbols and return symbols as proposed by VPLs [11], may be more practical. The different types of queries remain relatively simple (similar to the regular case) and providing the hierarchical information to the learner may be (technically) easier to implement on the SUL level rather than the query level. Especially for VPLs, [93, 109] provide positive results for the inference of these (instrumented) languages and the work of this thesis can be seen as a specialization of this line of thought.

8.4 Black-Box Checking and Learning-Based Testing

With the verification, testing, and learning of systems, the previous sections cover the related work on individual disciplines of MBQA. With black-box checking (BBC), Peled et al. [135] propose a concept for system verification that involves all three disciplines simultaneously. As the name suggests, BBC deals with the verification (or model-*checking*) of properties of black-box systems. Since black-box systems do not provide any reasonable model for the verification process, AAL and MBT may be used for the inference of such a model.

However, instead of executing these processes independently of each other, BBC proposes to exchange information between the model checker and the learning algorithm in order to enable a fruitful feedback loop. The model checker is used to verify properties on intermediate hypothesis models of the learner. If a property is violated, this may either be because the system does indeed violate the property or the hypothesis model is not yet accurate enough. The two cases can be distinguished by testing and the answer either solves the verification problem earlier (if the property is indeed violated by the system) or provides a counterexample for the learning algorithm (if the hypothesis model violates the property but the SUL does not). Here, the properties to be verified on the system serve as a guided counterexample search, which is particularly useful given that the general black-box equivalence problem is impossible to solve (cf. Section 2.3).

On the basis of the results of Chapters 4 to 6, one may implement a similar BBC workflow for (instrumented) context-free systems as well. Particularly for SPAs, SBAs, and SPMMs, Section 9.4 discusses some additional relationships between the individual processes, that make BBC especially fruitful for the presented model types.

Another approach to combine multiple individual disciplines of MBQA is that of *learning-based testing (LBT)* by Meinke and Sindhu [123, 124]. Instead of using AAL like BBC, LBT uses *passive automata learning* for constructing models of the black-box system. By aggregating potential counterexamples from model verification, LBT constructs a set of test cases that are evaluated on the black-box system via testing. Similar to BBC, these test cases either disprove a property directly or (in case of a false negative or false positive) uncover new behavior of the system. However, in LBT, these test cases (and the responses of the system to them) are used as training data for a passive learning algorithm and there exists no active learner that explores the system autonomously. Here, the learning process is more property-centric as any behavior irrelevant to the properties is not considered for model construction. Regarding LBT, Section 11.2.2 briefly discusses how the concept may be applied to SPAs, SBAs, and SPMMs.

CHAPTER 9

Practical Application of Instrumented Context-Free Systems

This chapter elaborates on various scenarios for the practical application of systems of procedural automata (SPAs), systems of behavioral automata (SBAs), and systems of procedural Mealy machines (SPMMs). It discusses the technical aspects of instrumenting systems and showcases application domains in which the proposed model types integrate naturally. The discussions focus on the conceptual aspects of employing SPAs, SBAs, and SPMMs in practice. For a performative evaluation, see Chapter 10.

9.1 Instrumentation

Before discussing the (technical) aspects of instrumenting a system, note that this process is only mandatory for active automata learning (AAL). If one is not interested in the hierarchical properties of a system, both the model verification process and the model-based testing (MBT) process can easily be adjusted to omit this information. The instrumented call symbols and return symbol can be simply filtered out from requirements and tests if these information are not relevant. Only AAL *requires* these semantics because it is the only process (in this thesis) that does not have access to a (white-box) model containing the structural information to begin with.

Providing an interface to a system that supports the proposed instrumentation in order to execute instrumented tests or queries may pose challenges. Specifically hardware systems may face the challenge of immutability which makes it hard to alter a system after construction. However, it can be argued that this specific problem is subsumed by the general problem of applying model-based quality assurance (MBQA) to hardware-based systems. Since most testing and learning tools are software-based, they also require a software-based interface to interact with hardware systems. This may either be achieved by corresponding adapters or virtualization, e.g., hardware description languages, which are effectively software systems again. As a result, the following discussions focus on the instrumentation of software systems.

For instrumenting software systems, there exists a large corpus of techniques and tools from the field of runtime verification (RV). RV faces a similar challenge of instrumenting systems in order to verify properties during execution, which allows one to utilize many of the existing solutions for this problem. See, e.g., [54] for a survey. Especially for

programming languages that use intermediate representations, such as the bytecode of the Java virtual machine (JVM) or the intermediate representation of the low-level virtual machine (LLVM), instrumenting a system is (technically) an easy process as concepts such as aspect-oriented programming allow one to inject the necessary instrumentation code prior to the execution on the actual hardware. Systems that are ahead-of-time compiled to native machine code, e.g., systems written in C, may be instrumented at link-time or execution-time using techniques such as dynamic binary instrumentation (DBI). See, e.g., tools such as [137] or [168].

If MBQA is used during development where the code of the system is available, one can also use code transformations, e.g., via pre-processors or compiler plugins, to incorporate the instrumentation during compilation. A lot of existing tools for code verification pursue this path (see, e.g., [19] for an example). In situations where the source-code is available, processes such as AAL become somewhat redundant because there is no longer a need for inferring a system model as it can be directly constructed from the source-code. However, the instrumentation may still be relevant if one is interested in verifying and testing the hierarchical properties of (instrumented) context-free systems. Furthermore, the control-flow graphs or data-flow graphs constructed from source-to-code transformers may be very verbose. In conjunction with manually defined input symbols (which play an important role for the degree of abstraction), applying AAL in scenarios where the source-code is available may still have its benefits.

In conclusion, implementing the proposed instrumentation requires some additional effort but it is in general not an obstacle for employing the proposed concepts in practice as there exist plenty of techniques and tools to support the necessary modifications in various scenarios.

9.2 Document Modeling

Besides the active modification of systems to incorporate the instrumentation required by SPAs, SBAs, or SPMMs, there also exist application domains where this kind of structure is *natural*. Specifically for SPAs, a rather intriguing example of this situation is the modeling of documents whose structure resembles a *tag language*. A prominent and widely used example of such a tag language is the extensible markup language (XML).

XML documents consist of a series of (hierarchically nested) tags which give structure to the information that a document describes. For every *opening* tag there must exist a matching *closing* tag at some point later in the document. Tags may be enriched with attributes and plain text may be used between tags to represent unstructured information of the document.

By interpreting individual tags as procedures, the concept of opening tags and closing tags directly corresponds to the notion of call symbols and return symbols. As a consequence, it is possible to associate words of an SPA with instances of XML documents. This allows one to represent the structure of certain XML documents via an SPA and vice versa. An SPA-based interpretation of XML documents not only captures the syntactical properties of the documents, e.g., the well-matchedness of tags, but also allows one to

discuss semantic properties of documents by utilizing the previously discussed techniques of MBQA.

Note that in XML documents, every opening tag needs a matching closing tag with the same name, i.e., an `<abc>` tag needs to be followed by a matching `</abc>` tag at some point. In the context of SPA languages this means that there exist multiple return symbols (one for each call symbol) whereas SPAs only support a single return symbol. This discrepancy is easily addressed by a thin translation layer (or *mapper*) that maps actual XML tags to abstracted call symbols and the return symbol. In particular, this mapper can be stateful such that when mapping a return symbol it has access to the current nesting hierarchy. This means one can easily map, e.g., the word "$a \cdot b \cdot r \cdot r$" to `<a><b></b></a>` and vice versa. Using such a mapper also allows one to translate other language features of XML documents, such as tag-attributes or arbitrary contents within tags, to, e.g., internal alphabet symbols.

Especially in the context of the world wide web, these structured documents are often used for the exchange of data between servers and clients. For example, the simple object access protocol (SOAP) [160] is a widely used standard for web-services that is based on sending hypertext transfer protocol (HTTP) requests with XML-encoded payloads. With the concept of mapping, it is also possible to describe other tag-languages such as the JavaScript object notation (JSON) which is a commonly used format in the context of representational state transfer (REST)-ful web services. Therefore, the concept of document modeling covers a broad area of application and is highly relevant for many real-world applications.

The following (sub-) sections expand on the idea of SPA-based interpretations of XML documents and discuss some practical examples for this approach.

9.2.1 DTD Learning

[60] elaborates on the concept of learning document type definitions (DTDs) based on analyzing the behavior of a black-box XML document validator. The paper discusses a (fictional) e-commerce shop that receives transaction data, e.g., orders, in the form of XML documents which contain information such as the ordered items, customer information, et cetera. Listing 9.1 gives an example of such an XML document that represents a valid order of the e-commerce shop. Each transaction consists of several records (`<records>`, `<record>`) which contain information about the transaction itself (`<tInf>`, `<date>`, `<reference>`) as well as customer information (`<cInf>`) and the purpose or duration of their storage (`<trans>`, `<crm>`, `<adv>`, `<delDate>`, `<disclaimer>`, `<agreement>`). The internal (DTD-based) description of what orders the e-commerce shop would classify as valid is shown in Listing 9.2.

As previously discussed, words of an SPA language may be transformed into XML documents with the help of a translation layer (or mapper) that maps between symbols of an SPA input alphabet and elements of an XML document. As shown in [60], this mapper may be used to further fine-tune the degree of detail with which the XML document

Listing 9.1 (from [60])
An exemplary XML document representing a valid order of the e-commerce shop.

```
<records>
    <record id="123">
        <date>2018-05-10</date>
        <tInf>Order No. 3434-CBGAE-45</tInf>
        <reference>catalog:CBGAE-4566X</reference>
        <reference>db:0234.23423-2</reference>
        <cInf type="address">Otto-Hahn-Str. 14</cInf>
        <purpose>
            <trans/>
            <delDate>2018-05-17</delDate>
        </purpose>
        <cInf type="e-mail">user@example.org</cInf>
        <purpose>
            <crm/>
            <disclaimer>Until canceled</disclaimer>
        </purpose>
    </record>
</records>
```

Listing 9.2 (from [60])
A DTD-based description of valid orders.

```
<!ELEMENT records (record+)>
<!ELEMENT record  (date,
                   ((tInf, reference+) |
                    (cInf, purpose))+) >

<!ELEMENT date      (#PCDATA)>
<!ELEMENT tInf      (#PCDATA)>
<!ELEMENT reference (#PCDATA)>
<!ELEMENT cInf      (#PCDATA)>
<!ELEMENT purpose   ((trans, delDate)  |
                     (crm, disclaimer) |
                     (adv, agreement)) >

<!ELEMENT trans      EMPTY>
<!ELEMENT crm        EMPTY>
<!ELEMENT adv        EMPTY>
<!ELEMENT delDate    (#PCDATA)>
<!ELEMENT disclaimer (#PCDATA)>
<!ELEMENT agreement  (#PCDATA)>

<!ATTLIST record id CDATA #IMPLIED>
<!ATTLIST cInf type CDATA #REQUIRED>
```

structure should be analyzed. For the following example, the partitioning

$$\begin{aligned}\Sigma_{call} &= \{\texttt{records}, \texttt{record}, \texttt{purpose}, \texttt{cInf}\},\\ \Sigma_{int} &= \{\texttt{date}, \texttt{tInf}, \texttt{reference}, \texttt{trans}, \ldots, \texttt{id}, \texttt{type}\}, \text{ and}\\ r &= \{R\}\end{aligned}$$

is chosen. This allows for the introspection of the four procedures (tags, respectively) <records>, <record>, <purpose>, and <cInf> while treating the other elements as (internal) atoms. By using the mapper to translate SPA words to XML documents, the e-commerce shop directly serves as an implementation of a membership query oracle (MQO) by testing whether the shop accepts the (translated) documents as valid orders. Here, the system does not require any additional instrumentation.

The result of the AAL process is an SPA-based description of the document structure (DTD) that the e-commerce shops classifies as valid. Figure 9.1 shows the procedures of the learned SPA model. Besides the syntactical properties of the documents (well-matchedness, tag-names, etc.), the SPA-based interpretation of the structure of XML documents directly opens the way for verifying semantic properties as well. The use-case in [60] is motivated by the (at that time relatively new released) general data protection regulation (GDPR) [53] which requires companies that process user data to (among other things) precisely describe which data the company stores and for what purpose. Given the SPA of Figure 9.1, one easily sees how the techniques presented in Chapter 4 can be used to verify properties such as "Every recorded customer information (<cInf>) must be justified by a purpose" or "Every type of purpose must be accompanied by its respective approval" to make sure that the e-commerce shop adheres to the GDPR requirements.

The generalization of this process (or rather the DTD-/XML-specific MBQA approach) is summarized in Figure 9.2. Steps ❶ and ❷ cover the exploration phase of the AAL algorithm. Here, membership queries (MQs) (with the help of a mapper) correspond to actual XML documents whose validity (membership) is checked by the backend of the e-commerce shop. Steps ❸ and ❹ cover the verification phase of AAL, which may use concepts from (context-free) MBT (cf. Chapter 5) to search for counterexamples given the current hypothesis model. Eventually, the learning process finishes and returns a hypothesis model in form of an SPA (❺) which may either be subject to further context-free model checking (❻, cf. Chapter 4) or transformed into a corresponding DTD specification (❼) that may be subject to further verification as well (❽).

This example shows how an SPA-based MBQA process can be implemented with relative ease (only requiring a simple mapper) for a practical real-world application.

9.2.2 Document-Driven Process Verification

[166] extends the idea of Section 9.2.1 to the concept of *document-driven process verification*. The idea of this concept is to not only model inputs or outputs of systems as, e.g., XML, documents but also internal workflows. This concept is a powerful enabler for SPA-based MBQA because the proposed instrumentation is no longer a factor that needs to be incorporated externally but an internal part of the core semantics (given

Figure 9.1 (from [60])
An SPA-based representation of the document structure of accepted orders of the e-commerce shop. Sink states and corresponding transitions are omitted for readability. Note that the image is directly exported from the implementation of the SPA learner and therefore lacks the ⌢ markup for the procedural context of input symbols.

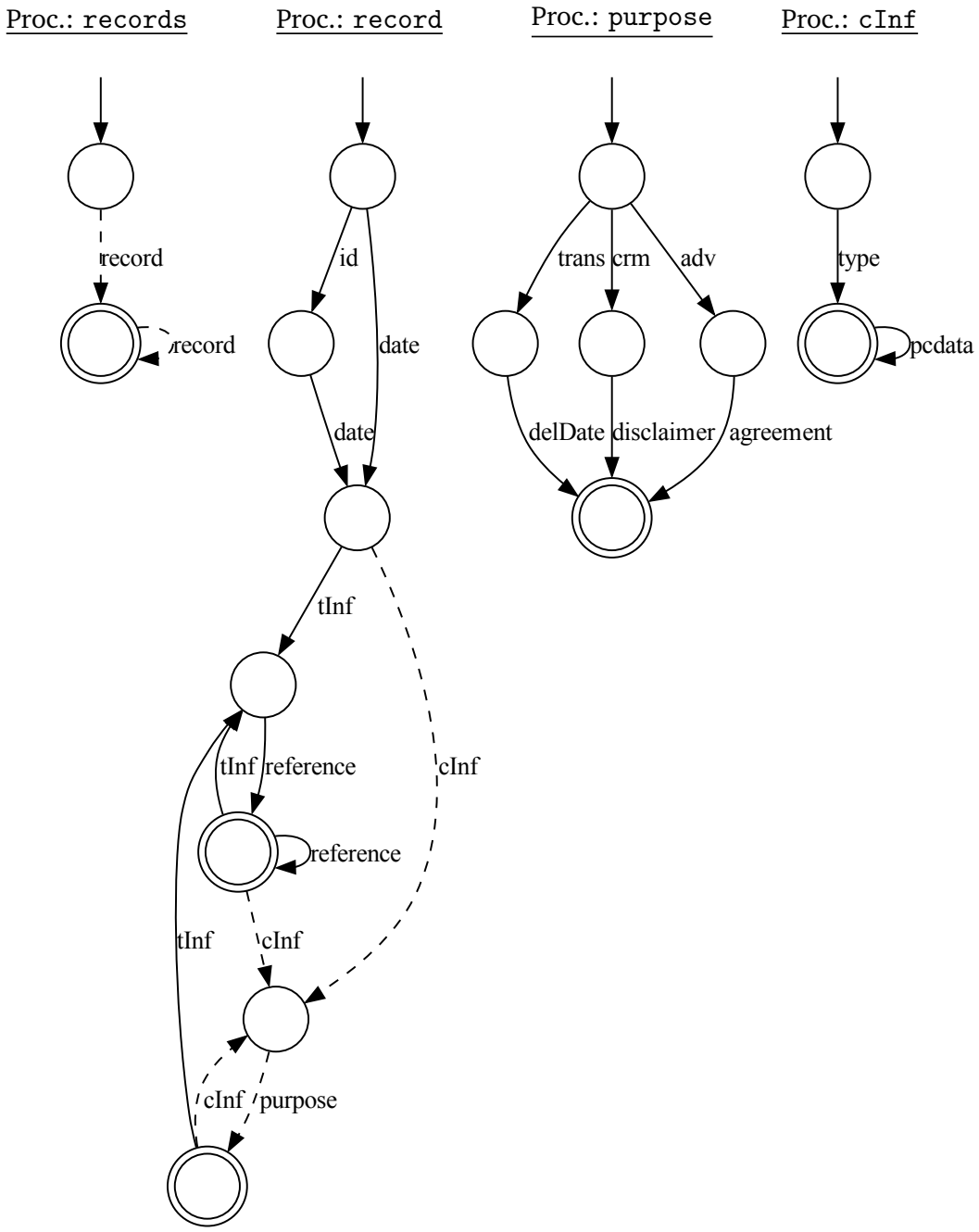
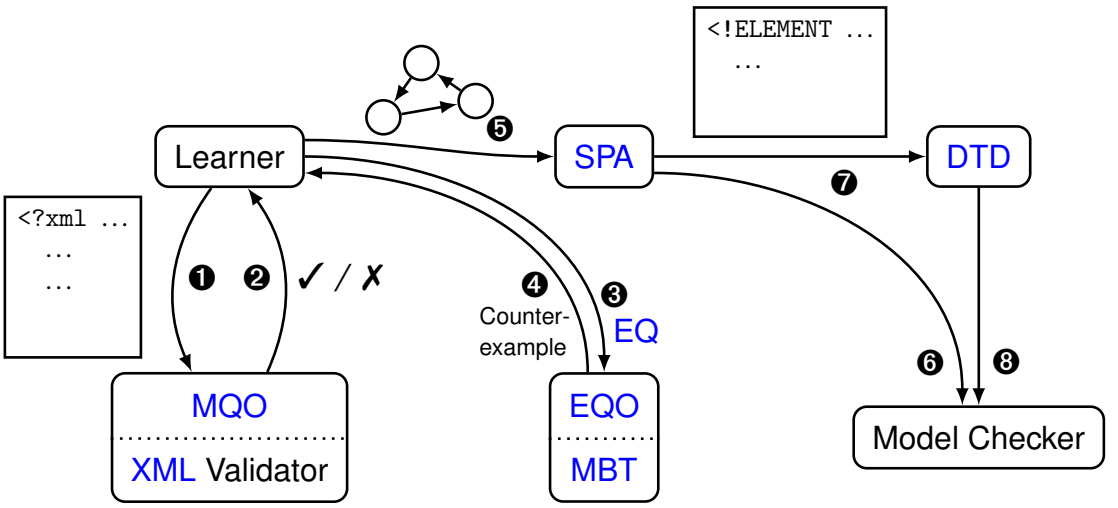


Figure 9.2 (from [60])
MBQA of DTD-based XML documents.



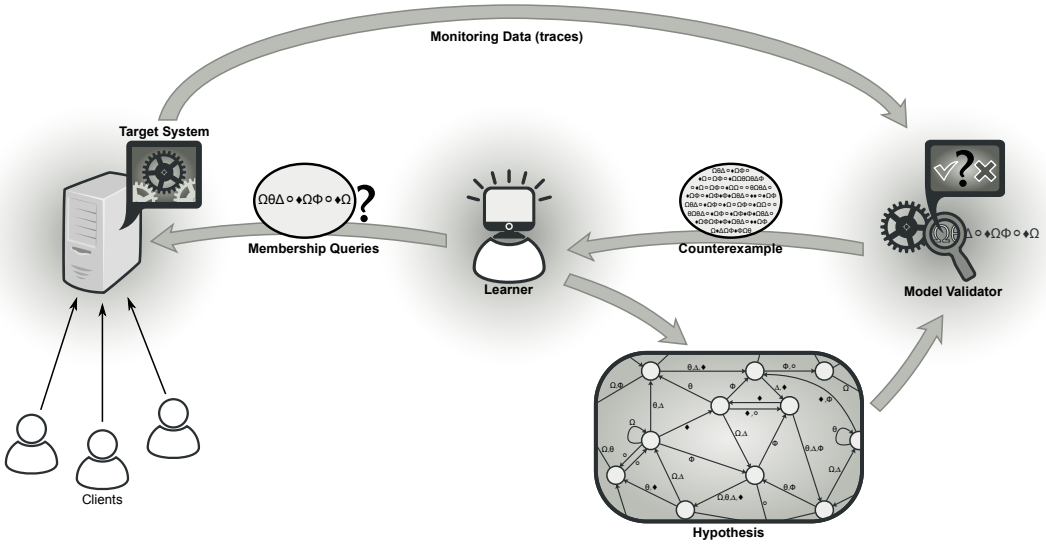
a fitting document type). Especially in large data-processing pipelines with multiple, micro-service-like components, documents allow for a convenient way to manage the global system complexity by using individual documents for individual components or processes. Furthermore, using documents for internal processing directly serves as a kind of logging framework which may be used for auditing, et cetera.

It is worth noting that in [166], this concept is elaborated in the context of whole product-line hierarchies, involving optional behavior in the form of (context-free) modal transition systems. However, the final products are always based on fully specified processes described by definitive documents, making this idea compatible with the concepts of Section 9.2.1.

9.2.3 XSD-Based Documents

Both, Section 9.2.1 ([60], respectively) and Section 9.2.2 ([166], respectively), only consider DTD-based XML documents. This is because DTD-based document generation directly combines the expansion semantics of context-free grammars (CFGs) with the proposed instrumentation of Definition 24 (via opening tags and closing tags), which nicely aligns with the semantics of SPAs, making the translation between the two formalisms very intuitive. However, with XML schema definitions (XSDs) there exists a strictly more powerful formalism than DTDs for specifying the structure of XML documents. Contrary to DTDs, XSDs allow one to specify the contents of a tag, i.e, the behavior of a procedure, depending on the context in which the tag is embedded. For example, given a tag c, an XSD is able to specify the two documents `<a><c>foo</c></a>` and `<b><c>bar</c></b>`. This is not possible with DTDs or SPAs. As a result, not all XSD-based XML documents can be described by SPAs.

Figure 9.3 (from [96])
The monitor-based "never-stop learning" approach, proposed by Bertolino et al. [26].



However, XSD-based XML documents can be fully described by visibly push-down automata (VPAs) [11]. Chapter 7 specifically discusses the differences and similarities between SPAs and VPAs and presents transformations between the two formalisms. These transformations allow one to construct SPA-based descriptions of XSD-based XML documents and apply SPA-based MBQA methods to these documents as well.

9.3 Monitoring and Life-Long Learning

One of the main challenges for AAL in practice is the search of counterexamples. As discussed in Section 2.4, the black-box equivalence problem is, in general, impossible to solve. At the same time, counterexamples are the driving force in AAL as each counterexample triggers a hypothesis refinement which makes the inferred model more precise. A particular interesting approach to tackle this challenge is that of monitor-based *never-stop learning* [26] or *live-long learning* as depicted in Figure 9.3. The following sections discuss the results of [59, 63] which apply this concept to instrumented context-free systems.

9.3.1 Monitoring

The main idea of this approach is to augment the system under learning (SUL) with a monitoring mechanism that is able to track interactions with and responses of the system. At first, the learner constructs a hypothesis model of the system via the classic learning loop, using conventional means of finding counterexamples. Then, if no more

counterexamples can be found and the application appears to be functional, the system is put into a production environment where external clients now interact with the system. During this time, a monitor records the interactions with the system and compares the recorded traces with the expected behavior of the hypothesis model.

If at one point the monitor detects a discrepancy between the recorded behavior and the behavior of the model, one of the following two situations have occurred:

1. The hypothesis behaves correctly and the system behaves faulty. In this case, the monitor has detected a bug in the system and the recorded trace can be used to reproduce the error and eventually fix the bug in the system.
2. The hypothesis behaves faulty and the system behaves correctly. In this case, the recorded trace represents a counterexample to the hypothesis model and the recorded trace can be used to refine the hypothesis model in a successive refinement step.

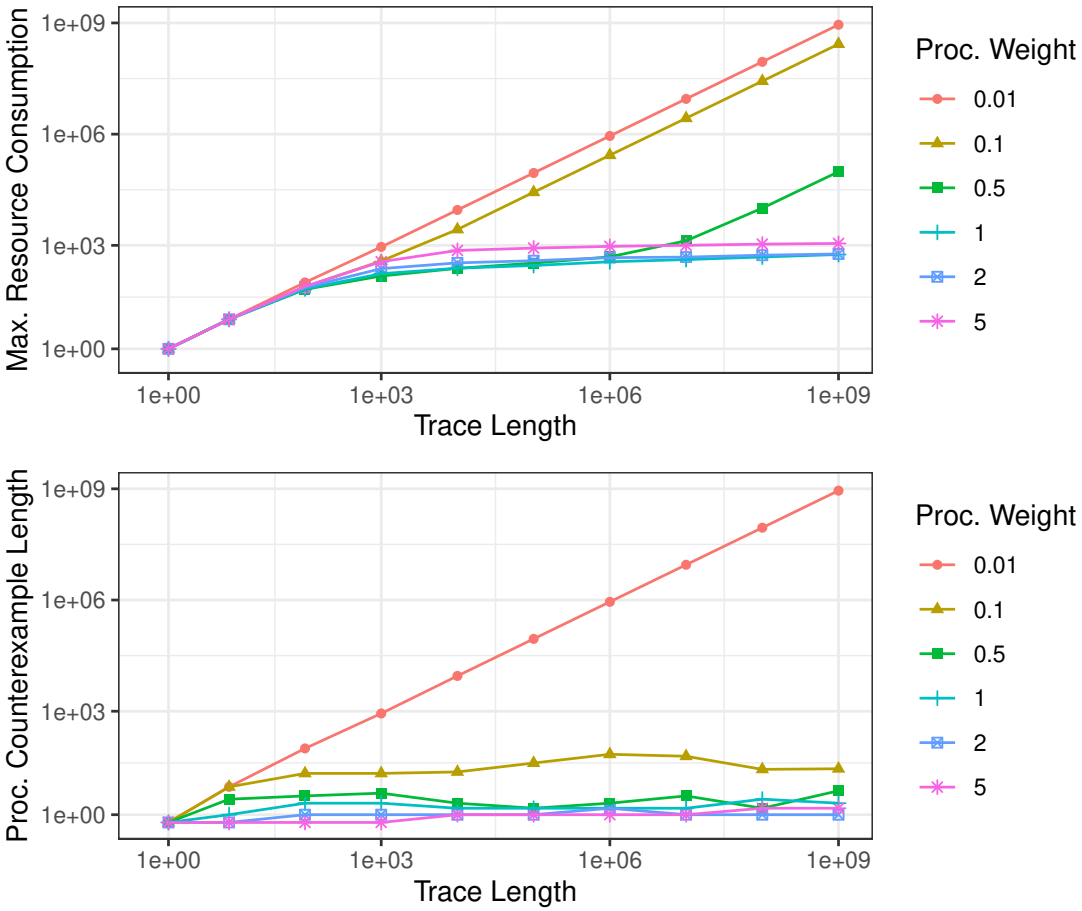
It is usually human resources, e.g., developers or quality assurance (QA) staff, who distinguish between the two cases, which is a common practice in the field of machine learning (cf. human-in-the-loop (HITL)).

One can think of this approach as a user-driven search for counterexamples which has shown great success in large-scale projects [5, 23, 25, 89, 99]. Especially in situations where systems are under-specified and therefore concepts such as model verification cannot be consulted for searching counterexamples, monitoring proves as a useful tool to observe and analyze the system from an external perspective.

In the context of instrumented context-free systems, [59] presents a notion of a (structural operational semantics (SOS)-based) monitor for SPAs. Essential to this monitor is the exploitation of the notion of rigorous (de-) composition of SPAs. As the refinement of SPA hypotheses essentially only requires projected counterexamples for the violating procedures (cf. Corollary 3), the monitor may perform these projections on-the-fly while simultaneously keeping track of the currently invoked procedure. As a result, when the monitor detects a mis-match in behavior, not only can it immediately provide a correct counterexample without any further analysis steps but also the resource consumption of the monitor is drastically reduced because the continuous projections of procedural runs allow the monitor to represent (potentially long but successful) invocations with a single call symbol.

Figure 9.4 shows an excerpt of the benchmark results of [59] that nicely show the impact of this concept. Even if the observed traces reach a length of a billion symbols, the maximum resource consumption of the monitor, i.e., the maximum number of symbols it needs to store at any time to successfully replicate the behavior, only reaches the thousands. Only for some corner-case scenarios where a system barely performs any procedural invocations (cf. procedural weights 0.01 and 0.1), the monitor needs to track all observed symbols due to the lack of possibilities to apply the proposed projection. As for the length of the extracted counterexamples, the second half of Figure 9.4 shows that (except for the corner-case scenarios again) the monitor is also able to extract efficient, i.e., short, counterexamples for the procedural learners, which further boost the performance of the learning process.

Figure 9.4 (from [59])
An excerpt of the (median) benchmark results of the SPA monitor.



The idea of reducing counterexamples can be found in other contexts that deal with (potentially infinite) domains as well. For example, in [3] the authors describe how the tool "Tomte" uses a pre-processing step to remove potentially redundant loops from expensive-to-analyze counterexamples for register automata. However, the proposed reduction is purely heuristic and requires a post-processing step to verify that the shortened counterexample is still valid. For instrumented context-free systems, the reduction via projections is an inherent semantically valid transformation that comes at no additional (query) costs.

The discussion in Section 3.5 shows that an SPA-based monitor is limited to verifying the termination of procedures. When observing monitor-friendlier reactive systems (modeled via, e.g., SBAs or SPMMs), the monitor may be simplified even further and the results shown in Figure 9.4 may be applied to an even broader field of systems.

9.3.2 Life-Long Learning

While a monitor for instrumented context-free systems can be implemented efficiently, it may still observe counterexamples only after days worth of observations, which makes them multiple orders of magnitude longer than "normal" counterexamples. [63] analyzes the impact of these monitor-based counterexamples on the learning process. In essence, counterexamples may contain a lot of inter-procedural redundancy, i.e., the redundancy of procedural invocations until the violating procedure is entered, and intra-procedural redundancy, i.e., the redundancy within a procedure until a violating action occurs, until the eventual cause of in-equivalent behavior is exposed. [63] shows that the combination of the proposed SPA monitor and SPA learner is able to perform well in these situations, too.

Remark 7
[63] calls inter-procedural redundancy external redundancy *and intra-procedural redundancy* internal redundancy.

For the inter-procedural redundancy, the notion of rigorous (de-) composition of SPAs allows for an efficient extraction of (local) counterexamples. As discussed in Section 9.3.1, the continuous projection of successfully terminated procedural invocations and the bookkeeping of the currently active procedure allow the SPA monitor to directly provide a projected procedural counterexample in case a mis-behaving action is detected. As a result, the global analysis for identifying the violating procedure (which is mainly affected by the inter-procedural redundancy) can be circumvented and the SPA learner may directly move to the procedural refinement.

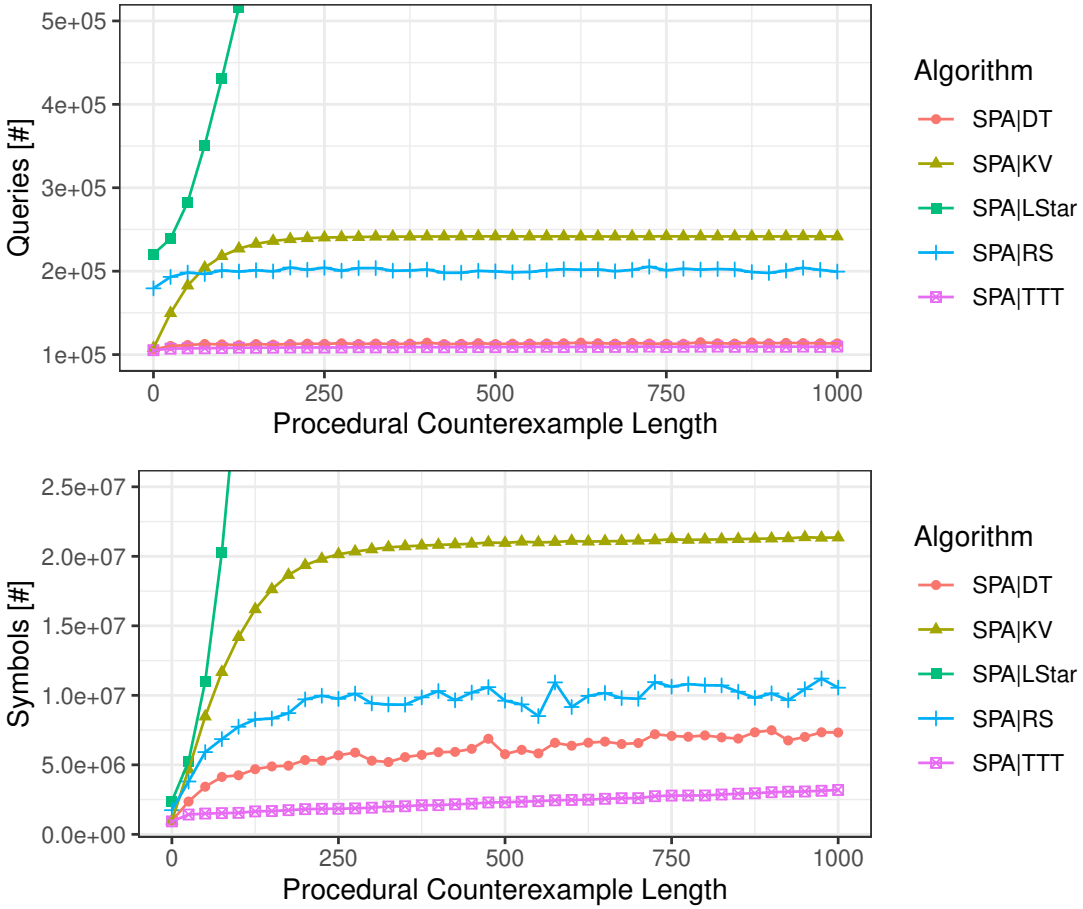
For the intra-procedural redundancy, the possibility to parameterize the SPA learner with arbitrary regular learners for the involved procedures allows one to transfer the properties of the (local) learners to the (global) learning process. As Figure 9.5 from [63] shows, algorithms such as the TTT algorithm [96] which specifically tackles the issue of redundancy within counterexamples allows one to drastically improve the query performance of the inference process compared to other configurations by better dealing with the intra-procedural redundancy within projected counterexamples. Here, the beneficial properties of its local learning behavior also positively affects the global learning behavior.

Overall, the benchmark results of [59, 63] show that the properties of SPAs allow for a fruitful application of monitor-based life-long learning of instrumented context-free systems. Similar, if not improved, results can be expected from SBAs and SPMMs as well.

9.4 Black-Box Checking and Other Symbioses

As presented in Section 8.4, black-box checking (BBC) describes the joint approach of verification, testing, and learning to boost the performance of the (black-box) model checking process. Besides the "classic" improvements found in this approach, BBC allows for some intriguing solutions to particular practical problems of SPA-based, SBA-based, and SPMM-based MBQA.

Figure 9.5 (from [63])
An excerpt of the benchmark results of different SPA learner parameterizations.



Recall from Chapter 6 that the SPA, SBA, and SPMM learners deal with the problem of missing access sequences, terminating sequences, and return sequences (in case of SPAs) via deferred learner activation. This concept culminates in the initial hypothesis which constitutes an empty SPA, SBA, or SPMM. Using only MBT for the search of counterexamples poses a challenge in this situation, as there exists no useful hypothesis model to generate test cases for. Even if the SPA, SBA, or SPMM hypotheses contain some non-empty procedural hypotheses, MBT still cannot cover procedures for which the corresponding regular learner has not yet been activated. Here, model verification comes as a remedy. By including requirements such as "the main procedure should successfully execute action a", the model checker finds that, e.g., the initial SPA hypothesis, violates this property and checks the SUL for confirmation. If the SUL satisfies this property, a positive counterexample is constructed that includes a successful invocation of the main procedure, activating the respective procedural learner. This way, by including similar

requirements for all involved procedures, model verification can be used to activate the respective procedural learners of the global learner and improve the performance of subsequent conformance tests. This is particularly fruitful application of the BBC concept, as requirements are normal inputs to this process and do not require any adjustments.

A similarly fruitful connection can be found between the learning and testing of models. While Chapter 5 presents a methodical approach for the conformance testing of the concerned model types, concepts such as the W-method often face practical problems due to the sheer amount of generated test cases. Especially for the counterexample search during AAL, this can be a performance concern. In practical challenges such as the ZULU challenge [48], promising results for the search of counterexamples have been shown by learning-based techniques [87]. Here, the intermediate data structures of learning algorithms, e.g., observation tables or discrimination trees (cf. Section 8.3), provide sensors for (states of) the SUL which are promising starting points to explore the system for in-equivalences. Since the inference of SPAs, SBAs, and SPMMs is based on the simultaneous inference of their respective regular procedures, a similar approach can be pursued for (instrumented) context-free systems, if the procedural learners support exposing the necessary information.

Note that for the successful exploration of the regular procedures on the global SUL, one requires access sequences, terminating sequences, and return sequences (in case of SPAs) similar to the conformance testing of models (cf. Chapter 5). Conveniently, the respective global learning algorithms already record the necessary sequences as they need them for query expansion as well. As a result, the same concept from regular learning and regular testing can be seamlessly lifted to the (instrumented) context-free case.

CHAPTER 10

Evaluation

This chapter discusses the qualitative and quantitative properties of systems of procedural automata (SPAs), systems of behavioral automata (SBAs), and systems of procedural Mealy machines (SPMMs) in comparison with competing formalisms. It summarizes the results of [61, 62, 63] and provides analyses to explain these results. Furthermore, this chapter analyzes the impact of the sequence optimizations of Section 6.2.5, which have not been discussed previously.

10.1 Qualitative Discussion

Discussing the qualitative aspects of a formalism is generally a challenging task because the perception of quality is often highly subjective. For example, often the size of models is used as a measure for the "complexity" of a formalism, where large models are considered complex and hard to understand. At the same time, having a low(er)-level and possibly more explicit representation of a system can make it easier to grasp the concrete properties of a system, which are otherwise only seen at a second glance. For the following discussion it should be noted that the arguments are based on the author's point of view and the reader may come to different conclusions.

The essential characteristic of SPAs, SBAs, and SPMMs is the notion of rigorous (de-) composition. Being able to (de-) compose a global, oftentimes highly complex, system (into) from individual and, more importantly, independent components is a key enabler for the analysis and understanding of systems. Similar to the criteria of Vardi [170] for specification formalisms (cf. Section 8.1), one may analyze the three formalisms regarding their applicability as model types.

Regarding expressiveness, SPAs are able to cover the whole set of (instrumented) context-free languages (CFLs) (cf. Theorem 3). SBAs add the notion of prefix-closure to a language. This not only covers prefix-closure of SPA languages but also includes new semantics such as non-terminating procedures which are not expressible via the SPA formalism. SPMMs add support for deterministic, symbol-wise transductions that follow an incremental lock-step pattern. While the class of context-free transductions strictly supersedes the class of SPMM-based transductions, e.g., by transductions of unequal input lengths and output lengths, SPMMs still provide an intuitive entry to context-free transductions. Overall, the three formalisms allow one the capture the core semantics of context-free (or procedural) systems.

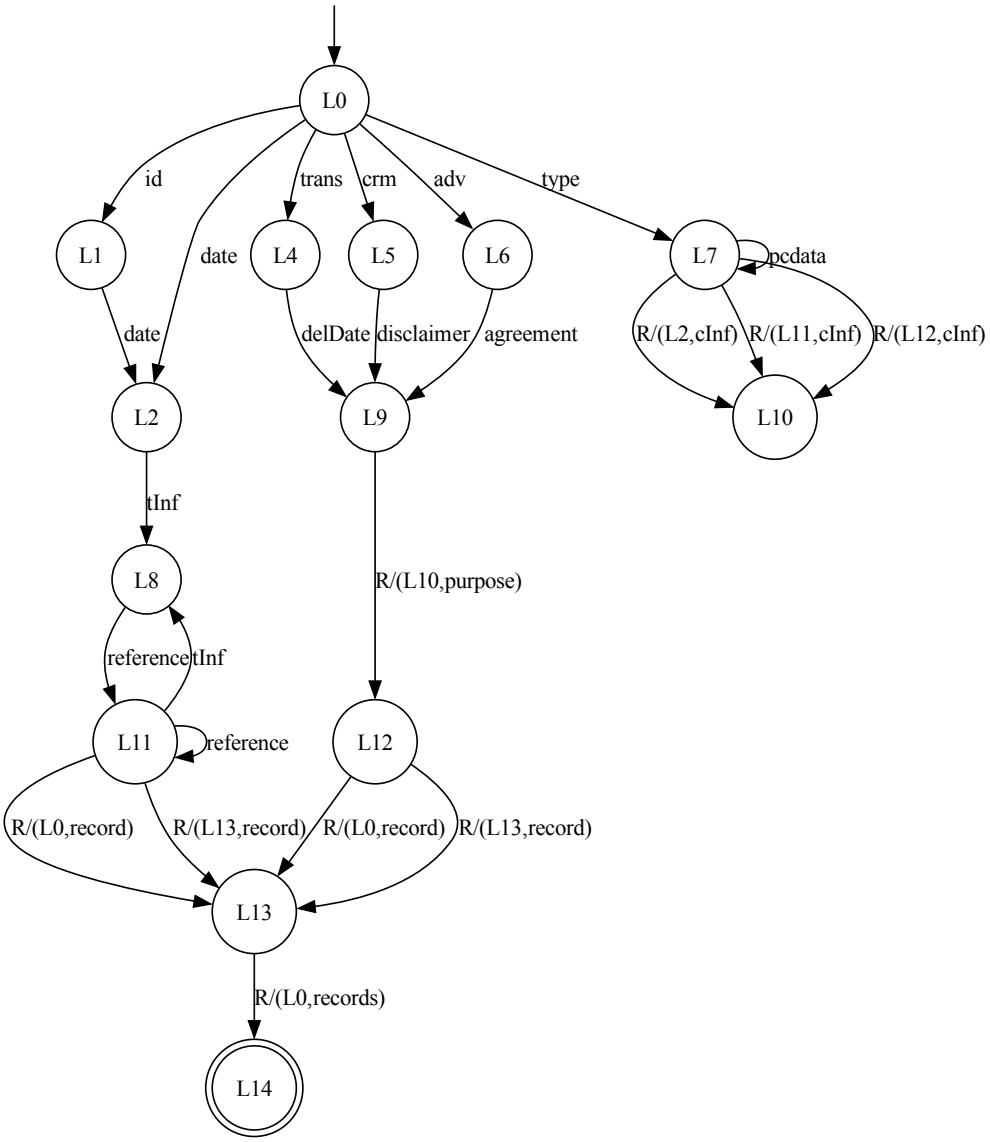
Regarding usability and complexity, SPAs, SBAs, and SPMMs benefit from the notion of rigorous (de-) composition as it allows them to represent their essential components, i.e., their procedures, via deterministic finite acceptors (DFAs) or Mealy machines. These procedural models types are simple, well-known, and their interaction which is based on the classic copy-rule semantics known from context-free grammars (CFGs) is easy to understand. The benefits of these properties become clear especially in a comparison with competing formalisms such as visibly push-down automata (VPAs). While VPAs support some notion of locality with the introduction of modules in the case of single-entry visibly push-down automata (SEVPAs), even the semantics of SEVPAs are still defined globally. The behavior of an individual module still needs information about the global context of a run in order to correctly determine, e.g., return transitions. Even for simple systems such as Figures 7.2 and 7.3 in Section 7.3, one easily sees the discrepancy in understandability. This effect becomes more apparent as the complexity of a system grows. Figure 10.1 shows the 1-SEVPA-based representation (or rather the "0-SEVPA"-based representation of [93], cf. Section 7.1.2) of the document type definition (DTD) model of the use-case discussed in Section 9.2.1. Comparing the model of Figure 10.1 with the SPA model based on Figure 9.1, one is able to make out certain areas with similar structure but especially the interactions between different procedures (calls to and returns from) are much more evident in the SPA-based representation. While VPAs offer, in general, more expressiveness, this property also comes at the cost of increased complexity.

Regarding composition, SPAs, SBAs, and SPMMs are the incarnation of this concept as the three formalisms are inherently defined as the composition of DFAs or Mealy machines. Extending any existing model is as simple as adding a new call symbol and a corresponding procedure (DFA or Mealy machine). The new procedure can be directly referenced by existing procedures via the respective call transitions. One of the major benefits of the notion of rigorous (de-) composition compared to, e.g., VPAs, is that the individual components (procedural automata, behavioral automata, or procedural Mealy machines) still have individual well-defined local semantics (that of DFAs or Mealy machines). This allows one to have both a global and a local view on the system, its components, and their respective behaviors. For example, from a bottom-up point of view, the simple and independent procedural components are aggregated to a greater, more complex, system. Similarly, from a top-down point of view, the functionality of the global system can be explained by various individual procedures that constitute the system.

Besides semantic properties, the notion of SPAs, SBAs, and SPMMs also offers benefits on a technical level. As formalisms for (instrumented) context-free systems, there exist different ways of implementing their semantics. This thesis presents stack-based characterizations that use a stack as the control component of the respective (language-) structural operational semantics (SOS) systems. Alternatively, they may be implemented using a CFG-based characterization or unrolling procedural invocations via graph transformation/rewriting [145]. The semantics of the proposed instrumentation and procedural systems are embeddable in various environments.

Furthermore, the idea of (de-) composition is extensible. As shown with SBAs and SPMMs, the core concept of SPAs can be easily extended with just minor adjustments

Figure 10.1
A 1-SEVPA representation of the document structure of valid orders of the e-commerce shop of the example of Section 9.2.1. Call transitions, sink locations, and corresponding transitions are omitted for readability.



such as including the return symbol in the procedural alphabet or considering an input alphabet over the cartesian product of some input domain and output domain. The core concepts of expansion and projection (and consequently the notion of rigorous (de-) composition) remains similar across all three formalisms. Adding additional properties may be implemented with similar ease and are briefly sketched in Chapter 11.

Concluding the qualitative discussion, SPAs, SBAs, and SPMMs provide a viable tool for the model-based quality assurance (MBQA) of (instrumented) context-free systems. The central enabler of these formalisms is the proposed instrumentation. While certain types of instrumentation, e.g., for VPAs, have fewer structural constraints and therefore allow for more general models (cf. Chapter 7), investing into a concise instrumentation as proposed in this thesis may offer qualitative and quantitative benefits later. Preempting the results of the quantitative discussion in Section 10.2, systems that support the proposed instrumentation yield smaller models and therefore perform better in the processes of MBQA such as active automata learning (AAL) (compared to VPAs). In situations where systems support the proposed type of instrumentation, having models in the form of SPAs, SBAs, and SPMMs, that support processes such as verification, testing, and learning, allows one to improve the MBQA experience and the success of its practical application.

10.2 Quantitative Discussion

For a quantitative discussion about the proposed model types, [61, 62] provide an initial comparison of SPAs and VPAs ([61]) as well as SPAs and SBAs ([62]) in the context of AAL. In order to elaborate on these results, Section 10.2.1 compares the model sizes of the different formalisms which directly provide an explanation for the observed data and an indication for the performance of other MBQA processes. Regarding SPAs and VPAs, the results of Chapter 7 allow for the transformation of SPAs into SEVPAs and vice versa, which makes it possible to analyze the respective models sizes for a given language. Regarding SPAs and SBAs, [62] shows by comparing the “classic” SPA-based learning to the combination of (prefix-closed) SBA learning and reduction (cf. Definition 48) that exploiting the available semantics of a system can improve the performance of the AAL process in certain situations. Here, this section continues this line of thought by comparing properties of SPMM-based representations of (instrumented) context-free transductions with their equivalent SBA-based representations in order to gauge the impact of the native model type.

Specifically in the context of AAL, [63] observes the fact that properties of regular learners transfer to the context-free context when using them as procedural learners for SPAs. A rather simple optimization heuristic therefore consists of using *efficient* regular learners as procedural learners. [63] shows how the notion of rigorous (de-) composition of SPAs allows one to tackle practical problems of AAL such as long counterexamples. Enabled by the same notion, Section 10.2.2 analyzes the impact of the sequence optimizations proposed in Section 6.2.5. The section elaborates on the potential performance improvements of this heuristic by comparing learning setups that use and do not use sequence optimizations.

All benchmarks use the implementations (learners, automaton types, etc.) available in version 0.16.0 of the open-source library LearnLib [95].

10.2.1 Models

For the comparison of SPAs and SEVPAs (and later SPMMs and SBAs), the benchmarks use synthetically generated model instances. While one may argue that these systems do not adequately represent real-life systems, synthetic systems allow for a more precise manipulation of their properties in order to gauge their impact on models and related processes. In contrast, real-life benchmarks (without any further analysis of systems properties) often only yield an individual data point without the ability to compare it to related data or systems.

SPAs as SEVPAs

For comparing the size of SPA models with the size SEVPA models, this comparison takes SPA languages and analyzes the SEVPA-based representations of these languages. Therefore, the benchmark starts with creating random SPAs, transforming them into a SEVPAs, and comparing the sizes of the models. The complete benchmark suite covers a series of twenty-five runs of which the averaged results are reported. We continue with looking at the details of a single run.

Generation For generating a random SPA, the benchmark first constructs a fixed SPA input alphabet Σ with four call symbols, ten internal symbols and the single return symbol, i.e., $|\Sigma| = 15$. Based on this alphabet, it generates four random procedural automata over $\widehat{\Sigma}_{proc}$ with $x \in \{2, 4, 8\}$ states each. For each procedural automaton and for each of its states, the successor states and acceptance are sampled according to a uniform distribution. The four procedural automata are then composed to an SPA according to four different configurations.

- Configuration 1 randomly selects one procedural automaton and uses it for all four call symbols, i.e., the SPA contains three duplicate procedures.
- Configuration 2 randomly selects two procedural automata and samples the remaining two procedures from the selected ones, i.e, the SPA contains two duplicate procedures.
- Configuration 3 randomly selects three procedural automata and samples the remaining one from the selected ones, i.e, the SPA contains one duplicate procedure.
- Configuration 4 uses all four procedural automata, i.e, the SPA contains no duplicate procedures.

For each configuration, the main procedure (call symbol) is selected at random as well. Note that the benchmark always checks that the generated SPA is minimal with respect to Σ (cf. Definition 30) and contains the necessary number of states, i.e., no procedural automata contain any equivalent states. If the SPA fails to meet these properties, the run restarts with a fresh set of randomly generated procedural automata.

Measurements For comparing the size of the SPA-based representation and the SEVPA-based representation, we look at the number of states (locations, respectively) and the number of transitions. Furthermore, the benchmark constructs a 1-SEVPA-based representation (or rather the "0-SEVPA"-based representation of [93], cf. Section 7.1.2) and a n-SEVPA-based representation. The 1-SEVPA-based representation unifies all locations in a single module and therefore does not duplicate equivalent locations across multiple modules. The n-SEVPA-based representation is more akin to the structure of SPAs as it contains one module per call symbol.

Results Table 10.1 shows the averaged results (including standard deviation) of twenty-five benchmark runs. The first observation is the fact that the size of the SPA models stays constant throughout all configurations. This was to be expected because the procedural automata are the generated source models of this benchmark suite and adhere to the chosen parameters. Therefore, they provide a reference for the size of the other model types.

The second observation concerns the size of the n-SEVPA-based models. Here, we see that the size remains consistent across all configurations as well. Similar to the SPA models, the n-SEVPA models describe each procedure in a separate module (set of locations) that does not share behavior with other modules. We see a slightly higher state (location) count because the transformation of Section 7.2 introduces a two-location main module that calls the initial procedure and for each module a sink-location is added with which rejected words transition into the main module again. Hence, the overhead is linear in the number of call symbols, i.e., $2 + |\Sigma_{call}|$. The increase in the transition count is more notable. This is due to the fact that the amount of (return-) transitions depends (in part) on the size of the stack alphabet which in case of SEVPAs is given by the product of all locations (across several modules) and call symbols. In contrast, procedural automata are only affected by the number of their local states and the size of $\widehat{\Sigma}_{proc}$.

The most notable observation can be made for the 1-SEVPA models. For the first configuration which exhibits a lot of similarity between the procedures, the 1-SEVPA models allow for the most compact representations of the languages, beating both the SPA models and n-SEVPA models. For the smallest systems ($x = 2$) this even holds true for the second configuration. However, with increasing diversity across the different procedures (i.e., increasing configuration number), the 1-SEVPA models fall victim to the combinatorial state explosion of handling up to four different procedural behaviors in a single module. Here, the data indicate that the 1-SEVPA models do not scale well with diversity across procedures. This is an interesting observation as it provides an explanation for the (in part) drastic performance differences between SPA and 1-SEVPA learning processes observed in [61, 63].

1-SEVPAs as SPAs

This comparison looks at the inverse direction and takes random well-matched visibly push-down languages (VPLs) in order to analyze the size of the SPA representations

Table 10.1
Sizes of SPA, 1-SEVPA and n-SEVPA models for (random) SPA languages.

Configuration		# of states / locations			# of transitions		
		$x = 2$	$x = 4$	$x = 8$	$x = 2$	$x = 4$	$x = 8$
SPA	1	8.0 ± 0.0	16.0 ± 0.0	32.0 ± 0.0	112.0 ± 0.0	224.0 ± 0.0	448.0 ± 0.0
	2	8.0 ± 0.0	16.0 ± 0.0	32.0 ± 0.0	112.0 ± 0.0	224.0 ± 0.0	448.0 ± 0.0
	3	8.0 ± 0.0	16.0 ± 0.0	32.0 ± 0.0	112.0 ± 0.0	224.0 ± 0.0	448.0 ± 0.0
	4	8.0 ± 0.0	16.0 ± 0.0	32.0 ± 0.0	112.0 ± 0.0	224.0 ± 0.0	448.0 ± 0.0
		$x = 2$	$x = 4$	$x = 8$	$x = 2$	$x = 4$	$x = 8$
1-SEVPA	1	3.0 ± 0.0	5.3 ± 0.5	9.6 ± 0.5	78.0 ± 0.0	188.6 ± 27.6	504.0 ± 45.0
	2	5.0 ± 0.0	17.7 ± 0.5	65.8 ± 0.4	170.0 ± 0.0	1504.9 ± 70.6	18240.4 ± 219.6
	3	9.0 ± 0.0	65.5 ± 0.7	509.2 ± 4.0	450.0 ± 0.0	18090.4 ± 349.9	1044164.6 ± 16178.4
	4	16.9 ± 0.3	251.1 ± 7.5	3842.3 ± 77.4	1382.3 ± 40.4	255977.4 ± 14886.4	59129284.9 ± 2364344.1
		$x = 2$	$x = 4$	$x = 8$	$x = 2$	$x = 4$	$x = 8$
n-SEVPA	1	14.0 ± 0.0	22.0 ± 0.0	38.0 ± 0.0	980.0 ± 0.0	2244.0 ± 0.0	6308.0 ± 0.0
	2	14.0 ± 0.0	22.0 ± 0.0	38.0 ± 0.0	980.0 ± 0.0	2244.0 ± 0.0	6308.0 ± 0.0
	3	14.0 ± 0.0	22.0 ± 0.0	38.0 ± 0.0	980.0 ± 0.0	2244.0 ± 0.0	6308.0 ± 0.0
	4	14.0 ± 0.0	22.0 ± 0.0	38.0 ± 0.0	980.0 ± 0.0	2244.0 ± 0.0	6308.0 ± 0.0

Table 10.2
Sizes of SPA models for (random) well-matched VPLs.

1-SEVPA						SPA								
$\lvert V\rvert$			$\lvert\delta\rvert$			$\lvert\breve{\Sigma}_{call}\rvert$			$\lvert\breve{S}\rvert$			# of transitions		
8.0	±	0.0	368.0	±	0.0	33.0	±	0.0	297.0	±	0.0	12771.0	±	0.0
16.0	±	0.0	1248.0	±	0.0	65.0	±	0.0	1105.0	±	0.0	82875.0	±	0.0
32.0	±	0.0	4544.0	±	0.0	129.0	±	0.0	4257.0	±	0.0	591723.0	±	0.0

of these languages. Therefore, similar to the previous (sub-) section, the benchmark generates random 1-SEVPAs and uses the concepts of Section 7.3 to construct language-equivalent SPAs. The benchmark focuses on VPLs described by 1-SEVPAs (or rather "0-SEVPAs" of [93], cf. Section 7.1.2), which allows for the use of existing functionality of LearnLib in order to execute the benchmarks. Note that this decision does not affect the concerned VPLs, as the choice of k for a k-SEVPA is only relevant for constructing a canonical representation.

Again, the complete benchmark suite covers a series of twenty-five runs of which the averaged results are reported. We continue with looking at the details of a single run.

Generation Similar to the previous section, the benchmark first constructs a fixed SPA input alphabet Σ with four call symbols, ten internal symbols and a single return symbol, i.e., $|\Sigma| = 15$. The benchmark generates a random SEVPA V over Σ with $x \in \{8, 16, 32\}$ locations. For each location, the internal successors, return successors, and its acceptance are sampled according to a uniform distribution. Recall that the successors of call transitions are predetermined by the module entry. Note that the benchmark always checks that the generated SEVPA is canonical and contains the necessary number of locations, i.e., no SEVPA contains two equivalent locations. If a SEVPA fails to meet this property, the run restarts with a new randomly generated SEVPA.

Measurements For analyzing the size of the SPA representation of the concerned VPL, we first look at the degree of refinement of the SPA, i.e, the number of required procedures to de-alias the different behaviors of the (abstract) call symbols. This number also indicates the size of the procedural alphabet, as we have $|\breve{\Sigma}_{proc}| =$ "# of procedures" $+ |\Sigma_{int}|$. Furthermore, we look at the number of states and the resulting amount of transitions of the procedural automata.

Results Table 10.2 shows the averaged results (including standard deviation) of twenty-five benchmark runs. When looking at the number of procedures, we see that the concretized SPAs require the maximum degree of refinement to describe the respective VPLs. We have $|\breve{\Sigma}_{call}| = 4x + 1$ for all sizes, where "$4x$" denotes the amount of location-specific call symbol refinements for each abstract call symbol of Σ_{call} and "+1" denotes the additional call symbol of the *main* procedure. Regarding the size of the procedural automata,

we see that they have to maintain the complete structural information of their originating modules. We have $|\breve{S}| = |\breve{\Sigma}_{call}| \cdot (x+1)$, where "+1" denotes an additional sink state in each procedural automaton. The amount of transitions is the product of $|\breve{\Sigma}_{proc}|$ and $|\breve{S}|$.

It is worth noting that for twenty-five benchmark runs the standard deviation is zero, i.e., every randomly generated SEVPA translates into an SPA with the maximum degree of refinement. The (likely) explanation for this effect is the fact that for any location in the main module of the SEVPAs there always exist paths to accepting locations, e.g., via internal symbols. Therefore, irrespective of the procedural behavior of the SEVPAs, each location can always reach an accepting location and therefore all procedural invocations are of relevance in the SPA representations. In this situation, the concretization merely resembles a cartesian unfolding of calls and locations. Here, SEVPAs are the more preferable representations for the concerned VPLs. However, randomly generated VPLs exhibit no inherent structure, which amplifies this effect. To get a more nuanced insight into the different representations, we continue to look at a second benchmark that concerns *constrained 1-SEVPAs*.

Constrained 1-SEVPAs as SPAs

For constraining the structure of VPLs, this comparison considers randomly generated SEVPAs which exhibit "procedural dead-ends". Again, the complete benchmark suite covers a series of twenty-five runs of which the averaged results are reported. We continue with looking at the details of a single run.

Generation This benchmark starts with constructing a SEVPA identical to the previous benchmark. Then, it randomly selects a (non-initial) location and transforms it into a sink. A sink s is a rejecting location such that all internal transitions and return transitions from s lead to s as well. Furthermore, all calls originating in s, i.e., return transitions where s is part of the top-of-stack tuple, also lead to s. This benchmark considers three different configurations in which it randomly selects x% (for $x \in \{25, 50, 75\}$) of the remaining (non-sink) locations and turns them into procedural dead-ends by updating all their return transitions to lead into s, i.e., returning in any of the sampled locations results in (global) rejection.

Measurements The benchmark collects the same data as in the previous one.

Results Table 10.3 shows the averaged results (including standard deviation) of twenty-five benchmark runs. For the number of procedures, we see that it scales nearly linearly with the amount of dead-ends, i.e., the 25% configuration contains about 25% less procedures compared to the previous benchmark, etc. Here, we see that the size of the SPA representations highly depends on the ability of well-matched SEVPA runs to reach an accepting location. Any procedural invocations that can be discarded, *are* discarded by the concretized SPAs.

Table 10.3
Sizes of SPA models for (random) well-matched VPLs with procedural dead-ends.

	1-SEVPA		SPA		
	$\lvert V\rvert$	$\lvert\delta\rvert$	$\lvert\breve{\Sigma}_{call}\rvert$	$\lvert\breve{S}\rvert$	# of transitions
25%	8.0 ± 0.0	368.0 ± 0.0	22.3 ± 4.4	178.2 ± 35.4	5904.3 ± 1870.3
	16.0 ± 0.0	1248.0 ± 0.0	44.7 ± 6.3	714.9 ± 101.1	39702.4 ± 9989.5
	32.0 ± 0.0	4544.0 ± 0.0	93.0 ± 9.8	2976.0 ± 313.5	309477.1 ± 60702.6
50%	8.0 ± 0.0	368.0 ± 0.0	15.9 ± 4.8	127.0 ± 38.7	3467.2 ± 1649.2
	16.0 ± 0.0	1248.0 ± 0.0	28.7 ± 9.2	458.9 ± 147.7	19058.6 ± 10580.3
	32.0 ± 0.0	4544.0 ± 0.0	62.9 ± 12.0	2013.4 ± 382.4	151207.7 ± 51935.2
75%	8.0 ± 0.0	368.0 ± 0.0	8.5 ± 4.8	68.2 ± 38.4	1439.7 ± 1046.5
	16.0 ± 0.0	1248.0 ± 0.0	15.2 ± 6.5	243.8 ± 104.6	6811.5 ± 4145.0
	32.0 ± 0.0	4544.0 ± 0.0	34.1 ± 11.9	1091.8 ± 381.4	52535.0 ± 31579.1

A similar trend is seen for the total number of states of the procedural automata. We see that the number of states directly corresponds to the product of the number of locations of the SEVPAs and the number of procedures.[5] This allows one to draw two interesting conclusions: First, due to the introduction of a sink location in the SEVPAs, this sink is now also directly a state in the respective procedural automata and does not introduce additional states like in the previous benchmark. Second, the remaining structure of the modules still needs to be fully represented as there exist corresponding states for every location. The (likely) explanation for this effect is the fact that, similar to the previous benchmark, all locations of the modules are reachable from the module entry via internal transitions and therefore relevant for the SPA representations. Hence, any further restrictions on the internal transitions of modules ,ay potentially reduce the size of the SPA representations further.

Regarding the number of transitions, the reductions in size apply in a super-linear fashion rather than in linear steps of 25%, 50%, or 75%. This is due to the fact that the number of transitions scales multiplicative with the number of states and the number of alphabet symbols (which depends on the number of procedures) which each experience individual reductions.

However, even in the configurations with the largest amount of procedural dead-ends, the size of the SEVPA representations still notably outperforms the size of the SPA representations. While the benchmark shows that SPAs have the potential to scale with the variability of VPLs, they are (quantitatively) a more verbose formalism for representing the concerned native VPLs.

[5] The raw data shows this correspondence. The displayed averaged data introduces some errors due to rounding.

Interpretation

In summary, the benchmark results of the comparisons between SPAs and SEVPAs align with the impressions of the qualitative discussion (cf. Section 10.1). SPAs (and by extension SBAs and SPMMs) are tools for modeling (instrumented) context-free systems. For systems where these tools are applicable (by means of the proposed instrumentation), they provide efficient models that yield concise system representations for the processes of MBQA. However, the advantages of these model types are not universal and there exist situations where they are outperformed by existing approaches such as VPAs.

This leads to the question: "When to use which formalism?". Interestingly, this question shifts the spotlight from the model types to the instrumentation mechanism. By implementing the instrumentation proposed in this thesis (cf. Definition 24), one enables the application of SPA, SBA, and SPMM model types and their benefits (efficiency, understandability, etc.). Especially for use-cases where this type of instrumentation can be found naturally (cf. Chapter 9), this results in fruitful applications that can outperform currently existing approaches. For systems where no such instrumentation is possible, existing alternatives may provide (quantitatively) better results.

SPAs, SBAs, and SPMMs offer system developers an efficient model type if they are willing or able to provide the necessary environment. Furthermore, the central role of the instrumentation opens up interesting fields of future research. Being able to automate or automatically optimize the proposed instrumentation of a system may be a key enabler for SPA-based, SBA-based, and SPMM-based MBQA of (instrumented) context-free systems. Section 11.2.2 discusses some ideas in that direction.

SPMMs vs. SBAs

Section 3.4 introduces SPMMs as a native formalism for (instrumented) context-free transductions. However, the verification, testing, and learning of SPMMs is formalized on the basis of SBAs over the cartesian product of some input domain and output domain. This (sub-) section analyzes the size of "native" SPMM models and their equivalent SBA-based representations in order to elaborate on the potential performance improvements given by the specialized representations. Similar to the previous comparisons, the whole benchmark suite covers a series of twenty-five runs of which the averaged results are reported. We continue with looking at the details of a single run.

Generation The benchmark generates an SPMM in analogy to configuration "4" of the "SPAs as SEVPAs" benchmark (cf. Section 10.2.1). It starts with constructing an SPA input alphabet containing four call symbols, ten internal symbols and the single return symbol as well as an SPA output alphabet containing ten internal output symbols and the two procedural output symbols. Then, it generates four random procedural Mealy machines with $x \in \{2, 4, 8\}$ states each such that successors and outputs are sampled according to a uniform distribution. For each procedural Mealy machine, a (non-initial) state s is selected at random and transformed into a sink state, i.e, a state with reflexive transitions for all $a \in \widehat{\Sigma}$ with output $\widehat{\text{👎}}$. For the remaining call transitions of the procedural Mealy machine,

Table 10.4
Sizes of SBA models for (random) SPMM-based transductions.

SPMM		$I \times O$-SBA		$I \cdot O$-SBA	
$\lvert S_M \rvert$	# of trans.	$\lvert S_B \rvert$	# of trans.	$\lvert S_B \rvert$	# of trans.
8.0 ± 0.0	120.0 ± 0.0	12.0 ± 0.0	1500.0 ± 0.0	51.4 ± 2.1	1388.9 ± 57.3
16.0 ± 0.0	240.0 ± 0.0	20.0 ± 0.0	2500.0 ± 0.0	120.9 ± 3.1	3264.8 ± 85.0
32.0 ± 0.0	480.0 ± 0.0	36.0 ± 0.0	4500.0 ± 0.0	257.0 ± 5.6	6940.1 ± 152.4

a coin-flip decides whether they output 👍 or transition into s with output 👎. Similarly, all remaining return transitions of the Mealy machine are updated to transition into s with outputs randomly sampled from {👍,👎}. If these modifications introduce equivalent states in a procedural Mealy machine or the composed SPMM is no longer valid, the run restarts with new randomly generated procedural Mealy machines.

Measurements For evaluating the efficiency of an SPMM model, we look at the size of the generated SPMM, the size of the (behaviorally) equivalent SBA over the synchronous alphabet, and the size of the (behaviorally) equivalent SBA over the alternating alphabet. Note that Section 2.1.3 only briefly sketches the acceptor-based characterization of transductions using alternating input symbols and output symbols and Section 3.4 does not consider it due to all the corner-cases it introduces. The SBA model over the alternating alphabet is only meant as an additional data-point to better position the results of SPMM and the following paragraphs do not go into further details about the construction of such SBAs.

In order to get a better insight into the structural properties of the different SBA models, the size of the models is split into the number of states and the number of respective transitions.

Results Table 10.4 shows the averaged results (including standard deviation) of twenty-five benchmark runs. For the SPMM models, we see that the respective size matches the configured parameters.

For the number of states of the synchronous SBAs, we see that the SBAs have exactly four states more than the originating SPMMs. This is due to the fact that the sink states in the generated SPMMs may still be valid states in successful invocations of the procedure, e.g., the targets of successful return-transitions. As a result, the transformed SBAs need to introduce additional (actual) sink states that are used for, e.g., successors of return transitions, et cetera. Together with the non-existing variance, this indicates that the synchronous SBAs are structurally very similar to the original SPMMs. However, regarding the number of transitions, we see that the synchronous SPA input alphabet drastically increases the number of transitions of the SBAs by about an order of magnitude. This heavily impacts the performance of testing and learning these kinds of systems and therefore favors the original SPMM models.

For the alternating SBAs, we see similar results. Here, the number of states drastically increases as each $\cdot \xrightarrow{a/o} \cdot$ transition in the SPMMs introduces additional states in the corresponding SBAs due to alternating $\cdot \xrightarrow{a} \cdot \xrightarrow{o} \cdot$ transitions. While the number of outgoing transitions per state is not as high as for the synchronous SBAs, we see that the total number of transitions begins to outnumber the synchronous versions with increasing number of states. Additionally, we start to see some variance in the data. This is mainly due do the fact that the successors of input-labeled transitions may be merged if the subsequent output-labeled transitions can separate the transduction steps again. Depending on the actual modeled transductions, this happens more often or not. However, given the total number of states, the impact of this effect is not sufficient for making alternating SBAs a competitive formalism.

Overall, these results support the efforts of formalizing a native Mealy-based formalism for modeling deterministic transductions that follow an incremental lock-step pattern. While the actual performance impact may differ from case to case, e.g., depending on the number of output symbols used, this benchmark shows that exploiting native system semantics such as dialog-based interactions, can result in a relatively great reduction in model size, which improves comprehensibility and performance — two important properties of practical MBQA.

10.2.2 Active Automata Learning

This section analyzes properties of SPAs in the context of AAL. In particular, we look at the optimizations of access sequences, terminating sequences, and return sequences discussed in Section 6.2.5. In AAL, algorithms are often analyzed by their query complexity, i.e., the number of membership queries (MQs) (and the number of symbols therein) that a learner poses during the inference process. On the one hand, this allows one to abstract from technical details, such as hardware platforms or programming languages, and compare different approaches on an algorithmic and conceptional level. On the other hand, for many practical applications the effective runtime of the inference process is determined by the performance of the system under learning (SUL) [90], i.e., the query performance of a learning algorithm has the most impact on the actual runtime. Therefore, a particularly interesting question in the context of SPA, SBA, and SPMM learning is how much the proposed optimizations can boost the learning performance. The following benchmark investigates this question for the case of SPAs.

Optimization of Access Sequences, Terminating Sequences, and Return Sequences

This benchmark constructs learning processes that infer synthetic systems again. Similar to previous benchmarks, the complete benchmark suite covers a series of twenty-five runs of which the averaged results are reported. We continue with looking at the details of a single run.

Generation For constructing the SUL that is inferred by the SPA learner, the benchmark uses the same system of configuration "4" of the "SPAs as SEVPAs" benchmark (cf. Section 10.2.1), i.e., an SPA consisting of four randomly generated procedural automata with $x \in \{2, 4, 8\}$ states over an SPA input alphabet containing four call symbols, ten internal symbols and the single return symbol.

Algorithms Recall that the SPA learner can be parameterized with arbitrary regular learning algorithms that are used as procedural learners. In each run, the benchmark infers the same system with four different procedural learning algorithms, namely:

- **LSTAR**, the original AAL algorithm by Angluin [15],
- **RS**, the LSTAR variant using the counterexample analysis of Rivest et al. [146],
- **DT**, the discrimination tree algorithm (sometimes called "observation pack" algorithm) by Howar [84], and
- **TTT**, the algorithm by Isberner et al. [96].

Counterexamples For generating counterexamples, the benchmark uses the SPA conformance test proposed in Definition 61. On the basis of the generated (white-box) SUL, it generates a set of test words that cover the characteristics of the system. The equivalence query oracle (EQO) then simply iterates over all these tests, executes them on both the current hypothesis model and the actual system, and returns the input sequence of a test as a counterexample if the results mismatch.

Additionally, the benchmark distinguishes between two configurations: It orders the test cases in lexicographical order and denotes as "short-to-long" the ascending order and as "long-to-short" the descending order. This distinction is meant to influence the extraction of the respective sequences of the SPA learner in order to gauge the impact of the potential improvements of the suggested optimizations.

Measurements For measuring the (query) performance of a learning setup, the following data are collected:

- **#EQ**, i.e., the number of equivalence queries (EQs) that have been posed during the learning process. This number minus one corresponds to the number of refinement steps each algorithm performs.
- **#MQ**, i.e., the number of MQs that the learning algorithm poses during hypothesis construction. This number *excludes* any MQs that are posed by the EQOs during the counterexample search but *includes* MQs posed by the (procedural) learners during counterexample analysis and hypothesis exploration.
- **#S**, i.e., the (cumulated) number of symbols of each of all recorded MQs.

Furthermore, a distinction is made between a "base" case which does not optimize the discovered access sequences, terminating sequences, and return sequences and an "optimized" case which replaces the respective sequences whenever a shorter one can be constructed.

Results Table 10.5 shows the averaged number of EQs, MQs and symbols (including standard deviation) of twenty-five benchmark runs. We first look at the number of EQs and the number of MQs. The overall observation is that there are nearly no differences between the base configuration and the optimized configuration. This was to be expected because the proposed optimizations only replace the respective sequences with shorter ones but maintain their characteristics of accessing, terminating, and returning from procedures. Occasionally, we can see a slight decrease in performance, e.g., in the number of MQs of the RS algorithm in the thirty-two-state system of the short-to-long configuration or in the number of EQs and MQs of the RS algorithm in the thirty-two-state system of the long-to-short configuration. This can be explained by the fact that for each refinement, the SPA learner must verify that the procedural automata are ts-conform with respect to the currently extracted terminating sequences (cf. Definition 65). Here, using shorter terminating sequences covers fewer transitions and therefore potentially discovers fewer inconsistencies directly. As a result, more global counterexamples may be needed to trigger the procedural refinements. However, as the data shows, this impact is mostly negligible.

Other than that, we see performance characteristics similar to the ones from regular language inference processes. On average, the LStar algorithm poses the most MQs due to its internal management of the observation table. As a result, the algorithm more thoroughly explores the system by itself and requires fewer counterexamples, i.e., EQs. In contrast, the discrimination tree-based algorithms (DT and TTT) require fewer MQs but more EQs. Furthermore, we observe the general trend that with increasing system size, the number of EQs and MQs also increases, which was to be expected. It is worth noting that for the smallest system configuration, all learners require the same amount of MQs. Recall that for the SPAs with eight states and four procedures, each procedure only consists of two states[6] which are discovered as soon as the learner is initialized. In the short-to-long configuration this means that a learner only needs one EQ to activate each procedural learner and a final one to determine equivalence. In the long-to-short configuration, a longer counterexample can uncover more than one procedure, which results in a total EQ count of below five.

The most notable impact of the proposed optimizations appears in the amount of (cumulated) symbols. First of all, we see a decrease in symbol count from the base case to the optimized case in all configurations, even in the ones where the optimizations result in an increase of MQs. Since the optimizations come for free (cf. Section 6.2.5), it is almost always advisable to use them, except in cases where resets (and therefore an increase in the number of MQs) dominate the actual query runtime. Generally, the impact of the performance improvements increases with increasing system size as small improvements continue to aggregate over multiple MQs. For the short-to-long configuration, we see improvements of about two percent up to improvements of about ten percent. For the long-to-short configuration, we see improvements of about ten percent up to improvements of about twenty-five percent.

For the base case, the long-to-short configuration performs worse than the short-to-long configuration. This was to be expected, since longer counterexamples result in

[6] One accepting state, one rejecting state. There exist no other minimal two-state automata.

Table 10.5
Impact of access sequence, terminating sequence, and return sequence replacements on the query performance of the SPA inference process. The abbreviations *XY(Y)* are constructed from $X \in \{B = \text{base}, O = \text{optimized}\}$, $Y(Y) \in \{MQ, EQ, S = \text{symbol}\}$.

short-to-long counterexample order

$\lvert S\rvert$	Alg.	# BEQ	# OEQ	# BMQ	# OMQ	# BS	# OS
8	LStar	5.0 ± 0.0	5.0 ± 0.0	116.0 ± 0.0	116.0 ± 0.0	776.8 ± 52.9	759.8 ± 53.8
	RS	5.0 ± 0.0	5.0 ± 0.0	116.0 ± 0.0	116.0 ± 0.0	776.8 ± 52.9	759.8 ± 53.8
	DT	5.0 ± 0.0	5.0 ± 0.0	124.0 ± 0.0	124.0 ± 0.0	818.2 ± 56.4	801.2 ± 57.4
	TTT	5.0 ± 0.0	5.0 ± 0.0	162.0 ± 0.0	162.0 ± 0.0	1104.4 ± 76.0	1083.0 ± 76.8
16	LStar	10.6 ± 1.2	10.6 ± 1.2	863.8 ± 140.2	863.8 ± 140.2	7962.9 ± 1491.2	7335.1 ± 1438.5
	RS	11.3 ± 1.1	11.3 ± 1.1	638.9 ± 60.5	638.9 ± 60.5	5447.8 ± 847.5	4882.8 ± 670.6
	DT	12.4 ± 0.8	12.4 ± 0.8	528.3 ± 21.9	528.3 ± 21.9	4456.6 ± 522.2	4067.8 ± 354.8
	TTT	12.4 ± 0.8	12.4 ± 0.8	584.0 ± 24.6	584.0 ± 24.6	4947.1 ± 569.4	4532.8 ± 388.6
32	LStar	16.6 ± 1.7	16.6 ± 1.7	3105.9 ± 310.2	3105.9 ± 310.2	33095.8 ± 3961.1	30834.0 ± 3930.5
	RS	18.9 ± 2.0	18.9 ± 2.0	2257.3 ± 219.2	2262.0 ± 216.6	22040.8 ± 2269.4	19953.2 ± 2343.8
	DT	26.8 ± 1.1	26.8 ± 1.1	1583.5 ± 51.0	1583.5 ± 51.0	15322.0 ± 810.2	14161.8 ± 957.8
	TTT	26.8 ± 1.2	26.8 ± 1.2	1625.8 ± 60.9	1625.8 ± 60.9	15606.6 ± 835.0	14412.1 ± 976.4

long-to-short counterexample order

$\lvert S\rvert$	Alg.	# BEQ	# OEQ	# BMQ	# OMQ	# BS	# OS
8	LStar	3.9 ± 0.7	3.9 ± 0.7	116.0 ± 0.0	116.0 ± 0.0	814.9 ± 60.3	775.0 ± 59.1
	RS	3.9 ± 0.7	3.9 ± 0.7	116.0 ± 0.0	116.0 ± 0.0	814.9 ± 60.3	775.0 ± 59.1
	DT	3.9 ± 0.7	3.9 ± 0.7	124.0 ± 0.0	124.0 ± 0.0	857.4 ± 64.0	817.5 ± 63.0
	TTT	3.9 ± 0.7	3.9 ± 0.7	162.0 ± 0.0	162.0 ± 0.0	1153.0 ± 81.8	1102.8 ± 82.2
16	LStar	10.2 ± 1.3	10.2 ± 1.3	843.5 ± 132.8	843.5 ± 132.8	8170.6 ± 2239.4	7041.1 ± 1452.5
	RS	10.5 ± 1.4	10.5 ± 1.4	631.1 ± 74.2	631.1 ± 74.2	5216.2 ± 1101.6	4631.1 ± 730.3
	DT	11.9 ± 0.9	11.9 ± 0.9	530.6 ± 21.0	530.6 ± 21.0	4635.1 ± 831.8	4031.3 ± 407.3
	TTT	11.9 ± 0.9	11.9 ± 0.9	578.6 ± 22.7	578.6 ± 22.7	5008.0 ± 852.8	4401.4 ± 413.8
32	LStar	16.1 ± 1.5	16.1 ± 1.5	3077.1 ± 373.4	3077.1 ± 373.4	38910.1 ± 12109.9	30978.9 ± 5962.6
	RS	18.0 ± 1.9	18.1 ± 1.9	2217.0 ± 196.5	2217.1 ± 196.4	23285.7 ± 6399.2	19113.0 ± 3130.0
	DT	26.8 ± 1.4	26.8 ± 1.4	1594.4 ± 52.1	1594.4 ± 52.1	19258.8 ± 5338.7	14910.2 ± 1955.6
	TTT	26.8 ± 1.6	26.8 ± 1.6	1627.2 ± 67.5	1627.2 ± 67.5	19105.4 ± 4691.2	15016.7 ± 1723.1

longer extracted sequences. However, as we see in the optimized case, the proposed optimizations almost boost the symbol performance of the long-to-short configuration to the level of the short-to-long configuration, occasionally beating it. This effect is in part supported by the reduced number of EQs and MQs in the long-to-short configuration. Longer counterexamples contain more information and therefore, may trigger multiple refinements at once. The proposed optimizations allow one to benefit from this effect without being (fully) affected by the negative impact of long counterexamples (long sequences). This effect may even be more impactful in live-long learning scenarios.

Overall, this evaluation underlines one central aspect of SPAs: scalability through the notion of rigorous (de-) composition. Only because of the fact that procedures contribute independently to the behavior of an SPA, one is able to exchange access sequences, terminating sequences, and return sequences. In the context of AAL, this benchmark shows how this property easily enables a *free* (symbol) performance boost of up to almost twenty-five percent. Due to similar properties of SBAs and SPMMs, the same heuristic should also be able to boost the (symbol) performance of their respective inference processes. It is an interesting question for future research, whether the notion of rigorous (de-) composition allows for further optimizations.

10.3 Summary

This section concludes the chapter by summarizing its main results.

- While a qualitative evaluation is highly subjective, SPAs, SBAs, and SPMMs support various concepts that enable intuitive representations of systems and therefore offer an attractive formalism for context-free MBQA.
- Quantitatively, SPAs (and it is reasonable to assume SBAs and SPMMs as well) hit a sweet spot for appropriately instrumented systems. While SPAs support transformations into SEVPAs and vice versa, SPAs are a more compact representation for SPA languages, whereas SEVPAs are a more compact representation for VPLs that are not an SPA language.
- In the context of AAL, these benefits transpire to the practical application of MBQA. The properties of SPAs, SBAs, and SPMMs enable the application of optimization heuristics which (in part significantly) boost the performance of the concerned disciplines for free.

CHAPTER 11

Summary and Future Work

This chapter concludes this thesis by summarizing its major results and presenting an outlook on potential future research topics.

11.1 Summary

This thesis presents the three formalisms of systems of procedural automata (SPAs), systems of behavioral automata (SBAs), and systems of procedural Mealy machines (SPMMs). SPAs are an automaton-based formalism for describing procedural systems modeled after context-free grammars (CFGs), in which individual deterministic finite acceptors (DFAs) represent the production rules of the involved non-terminal symbols. Calls to procedures are implemented via similar expansion semantics and, therefore, SPAs are able to represent systems modeled after context-free languages (CFLs). SBAs extend this concept by the idea of prefix-closure which allows for further features such as non-terminating procedures. SPMMs introduce the concept of dialog-based interactions, providing an entry model to context-free transductions.

A central trait of all three formalisms is an instrumentation that makes entries to and exits from procedural invocations observable. Returning to Example 2, it is easy to see how there exist multiple alternatives to describing an identical palindrome system, e.g., with using only a single non-terminal symbol (procedure). The proposed instrumentation of this thesis makes these otherwise hidden architectural properties of systems observable and treats them as first-class citizens.

Furthermore, the proposed instrumentation enables a notion of rigorous (de-) composition. Similar to the model types, where the "global" model is comprised of multiple "local" components or procedures, the global observable behavior of a system can be interpreted in local contexts of the involved procedures as well. Via projection and expansion, this thesis shows how to exploit the proposed instrumentation in order to establish an equivalence between the globally observable behavior and the locally observable behavior.

Based on this equivalence, this thesis presents algorithms for the verification, testing, and learning of the above models types for implementing the model-based quality assurance (MBQA) of instrumented context-free systems. Specifically for testing and learning of models, the provided algorithms exploit the notion of rigorous (de-) composition by implementing the two processes via simultaneous testing and learning of the individual components, respectively. While a similar approach is possible for the verification of

systems, this thesis additionally presents an approach for global system verification which allows one to specify global system requirements more intuitively.

In a comparison with competing formalisms such as visibly push-down automata (VPAs), this thesis shows that SPAs (and by extension SBAs and SPMMs) hit a sweet spot for certain system structures or types of system instrumentation, respectively. This thesis shows that both formalisms can be transformed into each other (with some minor technical adjustments) and that there exist certain systems for which SPAs (SBAs, SPMMs) provide the more efficient representations and systems where existing approaches such as VPAs provide the more efficient representations.

Regarding the applicability of SPAs, SBAs, and SPMMs in practice, this thesis discusses the technical aspects of and solutions for implementing the proposed instrumentation that allow one to use the three model types in real-world MBQA processes. This also includes scenarios in which the proposed instrumentation occurs naturally such as (document type definition (DTD)-based) extensible markup language (XML) document verification. Especially for combined approaches such as black-box checking (BBC), the model types allow for fruitful connections between the individual disciplines of MBQA in which, e.g., learning can benefit from verification, and testing can benefit from learning. Furthermore, SPAs, SBAs, and SPMMs support various usage profiles such as full system descriptions, e.g., document-based business processes, or stream-based system interfaces, e.g., monitoring of behavior, which make them a versatile tool for various scenarios.

Concluding this thesis, SPAs, SBAs, and SPMMs provide intuitive formalisms for modeling the behavior of instrumented context-free systems. With the availability of algorithms, implementations, and tools for the verification, testing, and learning of these models, they provide a novel utility in the tool-box of MBQA for improving the applicability, quality, and success of quality assurance (QA) in practice.

11.2 Future Work

On the basis of the results shown in this thesis, there exist various directions for extensions and future research.

11.2.1 Extensions of Procedural Models

One promising direction for future research is the integration of new system semantics by extending the models to capture additional properties. In part, this concept is already applied in this thesis. While SPAs form the “base” formalism that introduces concepts such as projection, extension, and (de-) composition, model types such as SBAs and SPMMs can be seen as extensions that add properties such as prefix-closure and transductions while preserving the core principles of SPAs. The following sections present further possible extensions.

Generalized Procedural Mealy Machines

Section 3.4 presents SPMMs as a formalism for a specific class of (instrumented) context-free transductions. Key to the formalization of SPMMs is an instrumentation that assigns successful and erroneous output symbols to the procedural call and return transitions, which allows for the characterization of SPMM transductions as a prefix-closed language over the cartesian product of input symbols and output symbols. However, in a more generalized setting, one may want procedural actions to emit *arbitrary* outputs. Here, the current characterization via SBAs reaches a limit.

While call symbols can certainly support multiple outputs, e.g., by introducing distinct (cartesian) call symbols in the form of $\langle c_1, o_1 \rangle, \langle c_1, o_2 \rangle, \ldots$, this is not possible for return symbols. Using $\langle r, o_1 \rangle, \langle r, o_2 \rangle, \ldots$, would introduce *multiple* return symbols which SBAs do not support. Note that this problem is caused by the characterization of transductions via formal languages. The procedural Mealy machines used in SPMMs easily support individual output symbols for individual return transitions while only requiring a single return (input) symbol. In order to support multiple output symbols on the global system level, it is necessary to apply the presented input transformations of SPAs and SBAs to output words as well. By projecting and expanding the outputs of a system, it should be a relatively straightforward process to establish the same notion of rigorous (de-) composition for this generalized class of (instrumented) context-free transductions.

(Sub-) Sequential Transducers

In a similar fashion to generalized SPMMs, one may consider more general classes of transductions such as sequential transducers (STs) or subsequential transducers (SSTs) [24]. An ST can be thought of as a Mealy machine whose (transition-) output function does not emit single output symbols but words over the output alphabet. Every run of an ST then outputs the concatenation of words of the traversed transitions. This concept can be extended to, e.g., SSTs which include state output words as well.

Crucial for enabling new procedural model types is the possibility to integrate an instrumentation that allows one to isolate the procedural components. Specifically for transductions, the work on and results of SPMMs gives a reference as to how to implement such an instrumentation for other model types such as SSTs. In turn, establishing a similar notion of rigorous (de-) composition then directly allows one to lift existing approaches for the "base" model type to a procedural context. For example, Vilar [171] presents an active automata learning (AAL) algorithm for SSTs, which may be used for learning *systems of procedural SSTs* in a similar fashion to how regular Mealy learners are used to learn SPMMs. Implementations for the verification and testing of such systems follow analogously.

Orthogonal System Properties

The motivation of SPAs, SBAs, and SPMMs is to provide model types that allow one to capture essential hierarchical properties of systems in an intuitive fashion. There exists a

lot of research on model types that cover other, orthogonal system properties such as data flow, time, or probability (cf. Chapter 8). A particularly interesting question would be how these independent dimensions can be combined with an instrumentation that exposes the internal structure of a system. Specifically the combination of data flow, e.g., via register automata, and hierarchy allows for very concise models of software systems, as these two concepts are a core feature of many modern programming languages. Finding fruitful combinations of properties potentially opens up a huge area for future research.

Yet, the work on SPAs, SBAs, and SPMMs shows that integrating hierarchy to existing system properties does not invalidate prior work. It should be the goal to keep up the central notion of rigorous (de-) composition in order to apply the already existing research and algorithms for the verification, testing, and learning of procedural components. Similar to how this thesis lifts regular MBQA to the context-free level, integrating other system properties may involve only a thin translation or aggregation layer as well.

11.2.2 Extensions of Applications

Besides extending the expressiveness of models, another line of (future) research may involve improving the applicability of the proposed concepts.

Self-Adjusting Instrumentation / Instrumentation Learning

Chapter 10 compares (among other things) the two formalisms of SPAs and single-entry visibly push-down automata (SEVPAs). The conducted benchmarks show that both formalisms have their respective sweet spots. While native SPA languages are efficiently represented by SPAs but not by SEVPAs, native (well-matched) visibly push-down languages (VPLs) are efficiently described by SEVPAs but not by SPAs. The question of which formalism to use is strongly connected to the question of what instrumentation is available.

This question opens up an interesting dimension for future research. This thesis always assumes a fixed alphabet and a fixed instrumentation. A possible extension to this concept would be a dynamic or self-adjusting instrumentation that introduces new call symbols on demand. Similar to the idea of automated alphabet abstraction refinement (AAAR) [86] which already plays an important role in the AAL process of SBAs and SPMMs, such an approach could gradually refine call symbols as needed by the behavior of the system. Formalizing this idea as a separate learning process could enable intuitive SPA models where otherwise different models, e.g., SEVPAs, would have been used due to performance concerns.

Section 7.4.2 already discusses a similar idea. However, in Section 7.4.2 the "concretization layer" is inside the SPA learner while the system still uses "abstract" call symbols, resulting in a lot of overhead compared to native SEVPA learning. With a self-adjusting instrumentation, the concretization moves into the system (or an external translation layer) thereby changing the exposed behavior. Here, the system adjusts towards the proposed instrumentation of this thesis which may favor the proposed MBQA processes compared to, e.g., SEVPA-based ones (cf. Chapter 10).

In a related line of thought, it would be interesting to see whether effects similar to classic (regular) AAL could be observed in practice. Within the minimally adequate teacher (MAT) framework, the existence of equivalence queries (EQs) allows for the formalization of correctness properties and termination properties of learning algorithms. In practice, however, EQs can only be approximated or require additional knowledge about the system under learning (SUL). Yet, there exist several success stories about the successful application of AAL (cf. Section 8.3). Similarly for SPAs, SBAs, and SPMMs, the existence of a correct instrumentation allows one to reason about the construction of algorithms for the verification, testing, and learning of the model types. It is an interesting question for future research whether approximation or additional knowledge would allow one to compensate for, e.g., an unreliable instrumentation or whether, e.g., an adjustable learner similar to Section 7.4.2 could successfully handle such systems.

X-by-Construction / Design for X

A similar line of thought concerns the *automated* integration of an instrumentation. Movements such as model-driven development (MDD) (or model-driven engineering (MDE)) [157] pick up on the power of models and already use them as first-class citizens during system development. Rather than constructing models a posteriori, e.g., via manual modeling or AAL, models are used a priori as the basis of an application. Using code-generators, these models are transformed into other models, are enhanced with various aspects, and ultimately are generated to executable code that integrates into the final application. By specifying and validating properties of these transformers and code-generators, one is able to establish a notion of *correctness-by-construction* [74, 75], meaning that certain traits of the source models directly manifest in the generated code. Generalizing this concept to not only cover correctness but arbitrary, non-functional properties leads to the concept of *X-by-construction* [22].

A particularly promising example of applying this concept in the context of SPAs, SBAs, and SPMMs is the tool Cinco [129]. Cinco allows one to build domain-specific integrated modeling environments that can be used for MDD or MDE. Lybecait et al. [113] coin the term "Design for 'X'" which is the Cinco-specific approach of implementing X-by-construction. Especially in the context of this thesis, incarnations of this concept in the form of "Design for Learnability" or "Design for Testability" could include an automated integration of the proposed instrumentation or the generation of alphabet definitions in order to execute tests or queries on the final application. Since Cinco-products, among other things, support modeling formal business processes with expansion semantics similar to CFGs, SPAs, SBAs, and SPMMs appear to be a natural model type for the MBQA of Cinco-product-based applications.

Counterexample Search in Active Automata Learning

Chapter 6 skips the discussion of searching counterexamples due to the topic being out of scope of this thesis. While model-based testing (MBT) (cf. Chapter 5) and BBC (cf. Section 9.4) are often used as successful heuristics to implement the search for

counterexamples in practice, there still exists a lot of room for future research in this field. For example, in the context of regular AAL, Smetsers et al. [159] find that adding fuzzing to the classic MBT-based counterexample search may reduce the time of finding counterexamples. It would be interesting to see, whether similar effects can be observed when learning (instrumented) context-free systems, i.e, whether results from regular counterexample search transfer to the context-free level as well. The notion of rigorous (de-) composition and in what ways it can be exploited, e.g., local counterexample search versus global counterexample search, is an interesting topic for future research.

Passive Learning and Learning-Based Testing

This thesis presents AAL algorithms for the inference of SPAs, SBAs, SPMMs in the context of the MAT framework. For the learning of models, one may also consider alternatives such as *passive automata learning* which infers models from a fixed training set of annotated runs. The work on passive automata learning predates the work on AAL (see, e.g., [29, 71] for examples of early popular passive automata learning algorithms for regular systems) and still is a very important area of automata learning today for scenarios where only pre-recorded data is available.

In the context of AAL, the notion of rigorous (de-) composition allows one to infer SPAs, SBAs, SPMMs via a simultaneous inference of the involved procedures. Here, in analogy to the active case, a passive SPA (SBA, SPMM) learner could project runs from the global training set to runs of the individual procedures in order to construct local training sets. While the (passive) inference of the individual procedures is generally not a problem, the projection step becomes a challenge. Recall from Section 6.2.2 that negative counterexamples require an intricate analysis process (including additional queries to the SUL) in order to determine violating procedures. Similarly, if the global training set contains a rejected run, it requires an analysis step to correctly construct the local training sets of the involved procedures. However, in a passive learning scenario, there exists no possibility to further analyze negative samples for the procedure(s) that is (are) responsible for rejecting the run.

A potential solution to this problem is to consider hybrid approaches like in learning-based testing (LBT) where passive learning is used to construct hypotheses but an active component, e.g., a membership query oracle (MQO), is used to evaluate test cases. Here, the active component may be exploited to not only evaluate individual test cases but also project and construct local training sets. This would allow SPAs to provide a possible solution for context-free LBT, which in itself would be an interesting topic for future research. For model types like SBAs and SPMMs, this problem may be less drastic due to the notion of *reduced counterexamples* (cf. Definition 67). Here, ensuring reduced traces during the construction of training data allows one to later exploit this property for an easier construction of local training sets because only the last active procedure can be responsible for rejection.

11.2.3 Extensions of Transformations

With Chapter 7, this thesis presents transformations from the proposed formalism of SPAs to VPAs and vice versa. It is an interesting question for future research, whether there exist more such transformations to other formalisms. Motivated by the use-case of XML documents, Schwentick [153] presents a survey of different automaton types for tree-based structures which have an inherent link to the (instrumented) CFLs discussed in this thesis. Formalisms such as bottom-up tree automata (BTAs), top-down tree automata (TTAs), or tree-walking automata (TWAs) follow the approach of separating the structure of trees from the structure of the corresponding automaton models, which enables a separate traversal of both components. While the previous formalisms receive inputs in the form of *trees*, they relate to formalisms such as VPAs which receive *document*-based descriptions of the same input.

As summarized in [153], (arbitrary) BTAs and (non-deterministic) TTAs are equally expressive to SEVPAs (which are referred to as "depth-synchronous push-down automata"). Here, the class of *deterministic TTAs* may be of special interest as these automata are strictly less expressive than their non-deterministic version. Examples of trees that cannot be accepted by deterministic TTAs exhibit similar problems of SEVPA-based words that cannot be accepted by SPAs (without transformations). It is an interesting question for future research, whether a correspondence between SPAs and deterministic TTAs could be established. Such a link would allow one to use the verification, testing, and learning of SPAs as a document-based approach for the verification, testing, and learning of deterministic TTAs.

List of Acronyms

AAAR	Automated Alphabet Abstraction Refinement
AAL	Active Automata Learning
ACTL	Action-based CTL
BBC	Black-Box Checking
BNF	Backus-Naur Form
BTA	Bottom-up Tree Automaton
CFG	Context-Free Grammar
CFL	Context-Free Language
CFPS	Context-Free Process System
CTL	Computational Tree Logic
CTL*	Computational Tree Logic with Linear Time Assertions
DBI	Dynamic Binary Instrumentation
DFA	Deterministic Finite Acceptor
DTD	Document Type Definition
EBNF	Extended Backus-Naur Form
EQ	Equivalence Query
EQO	Equivalence Query Oracle
GDPR	General Data Protection Regulation
HITL	Human-In-The-Loop
HML	Hennessy-Milner Logic
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
KTS	Kripke Transition System

LBT Learning-Based Testing
LLVM Low-Level Virtual Machine
LTL Linear Temporal Logic
LTS Labeled Transition System

MAT Minimally Adequate Teacher
MBQA Model-Based Quality Assurance
MBT Model-Based Testing
MDD Model-Driven Development
MDE Model-Driven Engineering
MQ Membership Query
MQO Membership Query Oracle

NWA Nested Word Automaton
NWTL Nested Word Temporal Logic

PDS Push-Down System
PLTL Linear Temporal Logic With Past
PPG Procedural Process Graph

QA Quality Assurance

REST Representational State Transfer
RSM Recursive State Machine
RV Runtime Verification

SBA System of Behavioral Automata
SDL Simple Deterministic Language
SEVPA Single-Entry Visibly Push-down Automaton
SOAP Simple Object Access Protocol
SOS Structural Operational Semantics
SPA System of Procedural Automata
SPMM System of Procedural Mealy Machines
SST Subsequential Transducer
ST Sequential Transducer
SUL System Under Learning

TTA Top-down Tree Automaton
TWA Tree-Walking Automaton

VLTL Visibly Linear Temporal Logic

VPA Visibly Push-down Automaton
VPL Visibly Push-down Language
VPT Visibly Push-down Transducer

XML Extensible Markup Language
XSD XML Schema Definition

List of Algorithms

List of Figures

List of Listings

List of Symbols

Alphabets

I, O	An input alphabet, output alphabet, page 11.
I^n	The set of words over I of length n, page 11.
I^*	The set of words over I of arbitrary, finite length, page 11.
I^+	The set of words over I of non-zero, finite length, page 11.
Σ	An SPA input alphabet, page 31.
Σ_{act}	The set of active alphabet symbols during the learning process, page 88.
Σ_{call}	The call alphabet of an SPA input alphabet, page 31.
Σ_{cont}	The set of continuable input symbols of an SBA, page 79.
Σ_{cur}	The set of currently eligible symbols of an SPA input alphabet during the computation of access sequences, terminating sequences, and return sequences, page 72.
Σ_{fin}	The set of finished call symbols of an SPA input alphabet during the computation of access sequences, terminating sequences, and return sequences, page 72.
Σ_{int}	The internal alphabet of an SPA input alphabet, page 31.
Σ_{ncon}	The set of non-continuable input symbols of an SBA, page 79.
Σ_{proc}	The procedural alphabet of an SPA input alphabet, page 31.
$\bar{\Sigma}$	An instrumented SPA input alphabet, page 41.
$\bar{\Sigma}_{call}$	The call alphabet of an instrumented SPA input alphabet, page 41.
$\bar{\Sigma}_{proc}$	The procedural alphabet of an instrumented SPA input alphabet, page 41.
$\widetilde{\Sigma}$	A visibly push-down alphabet, page 111.
$\widetilde{\Sigma}_{call}$	The call alphabet of a visibly push-down alphabet, page 111.
$\widetilde{\Sigma}_{int}$	The internal alphabet of a visibly push-down alphabet, page 111.
$\widetilde{\Sigma}_{ret}$	The return alphabet of a visibly push-down alphabet, page 111.
$\check{\Sigma}$	A concretized SPA input alphabet, page 122.
$\check{\Sigma}_{call}$	The call alphabet of a concretized SPA input alphabet, page 122.
$\check{\Sigma}_{proc}$	The procedural alphabet of a concretized SPA input alphabet, page 122.
$\Sigma^\times$	A synchronous SPA input alphabet, page 57.
Ω	An SPA output alphabet, page 56.
Ω_{int}	The internal alphabet of an SPA output alphabet, page 56.

Contexts

$\widehat{\ }$	The procedural interpretation of symbols, words, etc., page 31.
$\widetilde{\ }$	The VPA-based interpretation of symbols, words, etc., page 111.
$\breve{\ }$	The concretized interpretation of symbols, words, etc., page 122.

Functions

$\lvert A\rvert$	The size of a DFA A, page 15.
$\lvert I\rvert$	The size of an alphabet I, page 11.
$\lvert M\rvert$	The size of a Mealy machine M, page 17.
$\lvert S\rvert$	The size of an SPA S, page 34.
$\lvert S_B\rvert$	The size of an SBA S_B, page 50.
$\lvert w\rvert$	The length of a word w, page 12.
α	The alpha projection, page 38.
β	The call-return balance, page 35.
γ	The gamma expansion, page 37.
κ	The abstract translation function, page 126.
ρ_w	The maximum well-matched suffix function for a word w, page 35.
$Exp(S)$	The expansion of an SPA S, page 41.
$Exp(iCFPS^S)$	The expansion of an induced CFPS of an SPA S, page 64.
LI_{M_i}	The location identification function of module M_i, page 126.
$R(P_B^c)$	The reduction of a behavioral automaton P_B^c, page 55.
$[\![\cdot]\!]$	The alpha-gamma transformation, page 92.
$[\![\cdot]\!]^*$	The generalized alpha-gamma transformation, page 93.

Languages

L	A formal language, page 12.
$L_1 \cdot L_2$	The concatenation of two formal languages L_1 and L_2, page 12.
$L(A)$	The language of a DFA A, page 15.
$L(G)$	The language of a formal grammar G, page 14.
$L(S)$	The language of an SPA S, page 33.
$L(S_B)$	The language of an SBA S_B, page 48.
$CM(\Sigma)$	The set of call-matched words over Σ, page 35.
$RM(\Sigma)$	The set of return-matched words over Σ, page 35.
$WM(\Sigma)$	The set of well-matched words over Σ, page 35.
$MCM(\Sigma)$	The set of minimally call-matched words over Σ, page 35.
$MRM(\Sigma)$	The set of minimally return-matched words over Σ, page 35.
$MWM(\Sigma)$	The set of minimally well-matched words over Σ, page 35.

Models

A	A DFA, page 15.
G	A formal grammar, page 14.

H	A hypothesis model, page 87.
$iCFPS^S$	An induced CFPS for an SPA S, page 64.
$iPPG^c$	An induced PPG for a procedural automaton P^c, page 64.
$iPPG^c_B$	An induced behavioral PPG for a behavioral automaton P^c_B, page 68.
$\mathbb{L}$	An LTS, page 18.
$\mathbb{L}_{DFA}$	A DFA-based LTS, page 19.
$\mathbb{L}_{Mealy}$	A Mealy-based LTS, page 19.
M	A Mealy machine, page 17.
M_j	The module j of a SEVPA, page 113.
P^c	A procedural automaton for procedure c, page 31.
$\bar{P}^c$	An instrumented procedural automaton for procedure c, page 41.
$\breve{P}^{c_{i,j}}$	A concretized procedural automaton for procedure $c_{i,j}$, page 122.
P^c_B	A behavioral automaton for procedure c, page 46.
P^c_M	A procedural Mealy machine for procedure c, page 56.
q^*_j	The designated entry location of module M_j of a SEVPA, page 113.
S	An SPA, page 31.
$\bar{S}$	An instrumented SPA, page 42.
$\breve{S}$	A concretized SPA, page 123.
S_A	An accepting SPA, page 88.
S_B	An SBA, page 46.
S_M	An SPMM, page 56.
S_R	A rejecting SPA, page 88.
V	A VPA, page 111.
$V_{\simeq_L}$	A $\simeq_L$-induced VPA. page 115.

Relations

$\equiv_{DFA}$	The equivalence relation between two DFAs, page 15.
$\equiv_{Mealy}$	The equivalence relation between two Mealy machines, page 17.
$\equiv_{SBA}$	The equivalence relation between two SBAs, page 50.
$\equiv_{SPA}$	The equivalence relation between two SPAs, page 34.
$\simeq_L$	The unified congruence with respect to a well-matched language L, page 114.

Sets

Γ_{SPA}	The stack domain for SPAs, page 32.
Γ_{VPA}	The stack domain for VPAs, page 112.
AS_c	The set of access sequences for procedure c, page 52.
$Cont_c$	The context of a procedure c, page 36.
$CP(\widetilde{\Sigma})$	The set of context pairs over $WM(\widetilde{\Sigma})$, page 114.
$CP_S(\Sigma)$	The set of S-induced context pairs over $WM(\Sigma)$, page 116.
$CS(\mathbb{L})$	The characterizing set for an LTS $\mathbb{L}$, page 24.
$CT(\mathbb{L})$	The conformance test for an LTS $\mathbb{L}$, page 22.
$EATR(S)$	The set of extracted access sequences, terminating sequences, and return sequences of an SPA S, page 75.

$EAT(S_B)$	The set of extracted access sequences and terminating sequences of an SBA S_B, page 78.
$Inst_w$	The instances set of the word w, page 36.
$Pref(w)$	The set of all prefixes of w, page 12.
$Reach_I(\mathbb{L})$	The set of states of an LTS $\mathbb{L}$ reachable by symbols of I, page 23.
$SCS_I(\mathbb{L})$	The state cover set over I of an LTS $\mathbb{L}$, page 23.
$ST(\Gamma_{SPA})$	The set of stack configurations for SPAs, page 32.
$ST(\Gamma_{VPA})$	The set of stack configurations for VPAs, page 112.
$Suff(w)$	The set of all suffixes of w, page 12.
$T(M)$	The transductions of a Mealy machine M, page 17.
$T(S_M)$	The transductions of an SPMM S_M, page 59.
$TCS_I(\mathbb{L})$	The transition cover set over I of an LTS $\mathbb{L}$, page 23.
$cTCS(P_B)$	The continuable transition cover set of a behavioral automaton P_B, page 79.
$nTCS(P_B)$	The non-continuable transition cover set of a behavioral automaton P_B, page 79.
TS_c	The set of terminating sequences for procedure c, page 36, 52.

Symbols and Words

ε	The empty word, page 11.
a	An arbitrary alphabet symbol, page 11.
c_0	The initial procedure of SPAs, SBAs, and SPMMs, pages 31, 46, 56.
ce	A counterexample, page 25.
r	The return symbol of an SPA input alphabet, page 31.
as_c	An access sequence for procedure c, page 37.
ts_c	A terminating sequence for procedure c, page 37.
rs_c	A return sequence for procedure c, page 37.
$u, v, w, \ldots$	Symbols and words, page 11.
$w \cdot v$	The concatenation of symbols and words, page 12.
$w[i]$	The i-th symbol of the word w, page 12.
$w[i, j]$	The sub-word of w, ranging from index i to j (inclusively), page 12.
$w[, i]$	The prefix of w, ending at index i (inclusively), page 12.
$w[i,]$	The suffix of w, beginning at index i (inclusively), page 12.
$\perp$	The bottom-of-stack symbol, page 32.
👍	The “success” symbol of an SPA output alphabet, page 56.
👎	The “error” symbol of an SPA output alphabet, page 56.

List of Tables

Bibliography

[1] Fides Aarts, Joeri de Ruiter, and Erik Poll. “Formal Models of Bank Cards for Free”. In: *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE Computer Society, 2013, pp. 461–468. DOI: 10.1109/ICSTW.2013.60 (cit. on pp. 4, 137).

[2] Fides Aarts, Julien Schmaltz, and Frits W. Vaandrager. “Inference and Abstraction of the Biometric Passport”. In: *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 6415. Lecture Notes in Computer Science. Springer, 2010, pp. 673–686. DOI: 10.1007/978-3-642-16558-0_54 (cit. on pp. 4, 137).

[3] Fides Aarts et al. “Algorithms for Inferring Register Automata - A Comparison of Existing Approaches”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 8802. Lecture Notes in Computer Science. Springer, 2014, pp. 202–219. DOI: 10.1007/978-3-662-45234-9_15 (cit. on p. 150).

[4] Fides Aarts et al. “Automata Learning through Counterexample Guided Abstraction Refinement”. English. In: *FM 2012: Formal Methods*. Ed. by Dimitra Giannakopoulou and Dominique Méry. Vol. 7436. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 10–27. DOI: 10.1007/978-3-642-32759-9_4 (cit. on p. 137).

[5] Fides Aarts et al. *Establishing basis for learning algorithms*. Tech. rep. Feb. 2010. URL: http://hal.archives-ouvertes.fr/inria-00464671/en/ (cit. on p. 149).

[6] Fides Aarts et al. “Generating models of infinite-state communication protocols using regular inference with abstraction”. In: *Formal Methods Syst. Des.* 46.1 (2015), pp. 1–41. DOI: 10.1007/s10703-014-0216-x (cit. on pp. 4, 137).

[7] Rajeev Alur, Ahmed Bouajjani, and Javier Esparza. “Model Checking Procedural Programs”. In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 541–572. DOI: 10.1007/978-3-319-10575-8_17 (cit. on p. 134).

[8] Rajeev Alur, Swarat Chaudhuri, and Parthasarathy Madhusudan. “A Fixpoint Calculus for Local and Global Program Flows”. In: *SIGPLAN Notices* 41.1 (Jan. 2006), pp. 153–165. DOI: 10.1145/1111320.1111051 (cit. on p. 134).

[9] Rajeev Alur, Kousha Etessami, and Parthasarathy Madhusudan. “A Temporal Logic of Nested Calls and Returns”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 467–481. DOI: 10.1007/978-3-540-24730-2_35 (cit. on p. 134).

[10] Rajeev Alur and Parthasarathy Madhusudan. “Adding nesting structure to words”. In: *Journal of the ACM* 56.3 (2009), 16:1–16:43. DOI: 10.1145/1516512.1516518 (cit. on p. 134).

[11] Rajeev Alur and Parthasarathy Madhusudan. “Visibly Pushdown Languages”. In: *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*. STOC ’04. Chicago, IL, USA: Association for Computing Machinery, 2004, pp. 202–211. DOI: 10.1145/1007352.1007390 (cit. on pp. 111–113, 138, 148).

[12] Rajeev Alur et al. “Congruences for Visibly Pushdown Languages”. In: *Automata, Languages and Programming: 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005. Proceedings*. Ed. by Luís Caires et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1102–1114. DOI: 10.1007/11523468_89 (cit. on pp. 113, 115, 120).

[13] Rajeev Alur et al. “First-Order and Temporal Logics for Nested Words”. In: *Logical Methods in Computer Science* 4.4 (2008). DOI: 10.2168/LMCS-4(4:11)2008 (cit. on p. 134).

[14] Afshin Amighi et al. “Sound Control-Flow Graph Extraction for Java Programs with Exceptions”. In: *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*. Ed. by George Eleftherakis, Mike Hinchey, and Mike Holcombe. Vol. 7504. Lecture Notes in Computer Science. Springer, 2012, pp. 33–47. DOI: 10.1007/978-3-642-33826-7_3 (cit. on p. 3).

[15] Dana Angluin. “Learning Regular Sets from Queries and Counterexamples”. In: *Information and Computation* 75.2 (1987), pp. 87–106. DOI: 10.1016/0890-5401(87)90052-6 (cit. on pp. 4, 24, 136–138, 168).

[16] Dana Angluin. “Negative Results for Equivalence Queries”. In: *Machine Learning* 5 (1990), pp. 121–150. DOI: 10.1007/BF00116034 (cit. on p. 138).

[17] Dana Angluin and Michael Kharitonov. "When Won't Membership Queries Help?" In: *Journal of Computer and System Sciences* 50.2 (1995), pp. 336–355. DOI: 10.1006/jcss.1995.1026 (cit. on p. 138).

[18] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9 (cit. on pp. 3, 133).

[19] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. "A decade of software model checking with SLAM". In: *Commun. ACM* 54.7 (2011), pp. 68–76. DOI: 10.1145/1965724.1965743 (cit. on pp. 3, 142).

[20] Thomas Ball and Sriram K. Rajamani. "Bebop: A Symbolic Model Checker for Boolean Programs". In: *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings*. Ed. by Klaus Havelund, John Penix, and Willem Visser. Vol. 1885. Lecture Notes in Computer Science. Springer, 2000, pp. 113–130. DOI: 10.1007/10722468_7 (cit. on pp. 3, 135).

[21] Joachim Baran and Howard Barringer. "A Grammatical Representation of Visibly Pushdown Languages". In: *Logic, Language, Information and Computation, 14th International Workshop, WoLLIC 2007, Rio de Janeiro, Brazil, July 2-5, 2007, Proceedings*. Ed. by Daniel Leivant and Ruy J. G. B. de Queiroz. Vol. 4576. Lecture Notes in Computer Science. Springer, 2007, pp. 1–11. DOI: 10.1007/978-3-540-73445-1_1 (cit. on p. 113).

[22] Maurice H. ter Beek et al. "X-by-Construction - Correctness Meets Probability". In: *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 12476. Lecture Notes in Computer Science. Springer, 2020, pp. 211–215. DOI: 10.1007/978-3-030-61362-4_11 (cit. on p. 177).

[23] Amel Bennaceur et al. "Machine Learning for Emergent Middleware". In: *Trustworthy Eternal Systems via Evolving Software, Data and Knowledge - Second International Workshop, EternalS 2012, Montpellier, France, August 28, 2012, Revised Selected Papers*. Ed. by Alessandro Moschitti and Barbara Plank. Vol. 379. Communications in Computer and Information Science. Springer, 2012, pp. 16–29. DOI: 10.1007/978-3-642-45260-4_2 (cit. on p. 149).

[24] Jean Berstel. *Transductions and context-free languages*. Vol. 38. Teubner Studienbücher : Informatik. Teubner, 1979. DOI: 10.1007/978-3-663-09367-1 (cit. on pp. 9, 175).

[25] Antonia Bertolino et al. *Further development of learning techniques*. Research Report. Feb. 2011. URL: https://hal.inria.fr/inria-00584926 (cit. on p. 149).

[26] Antonia Bertolino et al. “Never-stop Learning: Continuous Validation of Learned Models for Evolving Systems through Monitoring.” In: *ERCIM News* 2012.88 (2012), pp. 28–29. URL: http://ercim-news.ercim.eu/en88/special/never-stop-learning-continuous-validation-of-learned-models-for-evolving-systems-through-monitoring (cit. on pp. 4, 6, 137, 148).

[27] Dirk Beyer and M. Erkan Keremoglu. “CPAchecker: A Tool for Configurable Software Verification”. In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 184–190. DOI: 10.1007/978-3-642-22110-1_16 (cit. on p. 3).

[28] Dirk Beyer et al. “The software model checker Blast”. In: *STTT* 9.5-6 (2007), pp. 505–525. DOI: 10.1007/s10009-007-0044-z (cit. on p. 3).

[29] Alan W. Biermann and Jerome A. Feldman. “On the Synthesis of Finite-State Machines from Samples of Their Behavior”. In: *IEEE Trans. Computers* 21.6 (1972), pp. 592–597. DOI: 10.1109/TC.1972.5009015 (cit. on p. 178).

[30] Benedikt Bollig et al. “A Fresh Approach to Learning Register Automata”. In: *Developments in Language Theory: 17th International Conference, DLT 2013, Marne-la-Vallée, France, June 18-21, 2013. Proceedings*. Ed. by Marie-Pierre Béal and Olivier Carton. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 118–130. DOI: 10.1007/978-3-642-38771-5_12 (cit. on p. 137).

[31] Benedikt Bollig et al. “libalf: The Automata Learning Framework”. In: *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. Ed. by Tayssir Touili, Byron Cook, and Paul B. Jackson. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 360–364. DOI: 10.1007/978-3-642-14295-6_32 (cit. on p. 137).

[32] Laura Bozzelli. “Alternating Automata and a Temporal Fixpoint Calculus for Visibly Pushdown Languages”. In: *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*. Ed. by Luís Caires and Vasco Thudichum Vasconcelos. Vol. 4703. Lecture Notes in Computer Science. Springer, 2007, pp. 476–491. DOI: 10.1007/978-3-540-74407-8_32 (cit. on p. 134).

[33] Laura Bozzelli and Ruggero Lanotte. “Hybrid and First-Order Complete Extensions of CaRet”. In: *Automated Reasoning with Analytic Tableaux and Related Methods - 20th International Conference, TABLEAUX 2011, Bern, Switzerland, July 4-8, 2011. Proceedings*. Ed. by Kai Brünnler and George Metcalfe. Vol. 6793. Lecture Notes in Computer Science. Springer, 2011, pp. 58–72. DOI: 10.1007/978-3-642-22119-4_7 (cit. on p. 134).

[34] Laura Bozzelli and César Sánchez. "Visibly Linear Temporal Logic". In: *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*. Ed. by Stéphane Demri, Deepak Kapur, and Christoph Weidenbach. Vol. 8562. Lecture Notes in Computer Science. Springer, 2014, pp. 418–433. DOI: 10.1007/978-3-319-08587-6_33 (cit. on p. 134).

[35] Julian C. Bradfield and Colin Stirling. "Modal mu-calculi". In: *Handbook of Modal Logic*. Ed. by Patrick Blackburn, J. F. A. K. van Benthem, and Frank Wolter. Vol. 3. Studies in logic and practical reasoning. North-Holland, 2007, pp. 721–756. DOI: 10.1016/s1570-2464(07)80015-2 (cit. on p. 21).

[36] Manfred Broy et al. *Model-Based Testing of Reactive Systems:* vol. 3472. Lecture Notes in Computer Science. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. DOI: 10.1007/b137241 (cit. on pp. 3, 21, 135).

[37] Olaf Burkart and Bernhard Steffen. "Model checking for context-free processes". In: *CONCUR '92*. Ed. by W.R. Cleaveland. Vol. 630. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1992, pp. 123–137. DOI: 10.1007/BFb0084787 (cit. on pp. 5, 63–66, 134).

[38] Olaf Burkart and Bernhard Steffen. "Model Checking the Full Modal Mu-Calculus for Infinite Sequential Processes". In: *Automata, Languages and Programming*. Ed. by Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela. Vol. 1256. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997, pp. 419–429. DOI: 10.1007/3-540-63165-8_198 (cit. on pp. 66, 134).

[39] Sofia Cassel, Falk Howar, and Bengt Jonsson. "RALib: A LearnLib extension for inferring EFSMs". In: *DIFTS* 5 (2015) (cit. on p. 137).

[40] Sofia Cassel et al. "Learning Extended Finite State Machines". English. In: *Software Engineering and Formal Methods*. Ed. by Dimitra Giannakopoulou and Gwen Salaün. Vol. 8702. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 250–264. DOI: 10.1007/978-3-319-10431-7_18 (cit. on p. 137).

[41] Chia Yuan Cho et al. "Inference and analysis of formal models of botnet command and control protocols". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*. Ed. by Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov. ACM, 2010, pp. 426–439. DOI: 10.1145/1866307.1866355 (cit. on pp. 4, 137).

[42] Noam Chomsky. "On Certain Formal Properties of Grammars". In: *Information and Control* 2.2 (1959), pp. 137–167. DOI: 10.1016/S0019-9958(59)90362-6 (cit. on p. 138).

[43] Noam Chomsky. "Three models for the description of language". In: *IRE Transactions on Information Theory* 2.3 (1956), pp. 113–124. DOI: 10.1109/TIT.1956.1056813 (cit. on p. 13).

[44] Tsun S. Chow. “Testing Software Design Modeled by Finite-State Machines”. In: *IEEE Transactions on Software Engineering* 4.3 (1978), pp. 178–187. DOI: 10.1109/TSE.1978.231496 (cit. on pp. 22, 71, 78, 80, 81, 135).

[45] Alessandro Cimatti et al. “NuSMV 2: An OpenSource Tool for Symbolic Model Checking”. In: *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*. Ed. by Ed Brinksma and Kim Guldstrand Larsen. Vol. 2404. Lecture Notes in Computer Science. Springer, 2002, pp. 359–364. DOI: 10.1007/3-540-45657-0_29 (cit. on p. 3).

[46] Edmund M. Clarke and E. Allen Emerson. “Design and synthesis of synchronization skeletons using branching time temporal logic”. In: *Logics of Programs*. Ed. by Dexter Kozen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 52–71. DOI: 10.1007/BFb0025774 (cit. on pp. 2, 19, 133).

[47] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, USA, 1999. ISBN: 9780262032704 (cit. on pp. 3, 133).

[48] David Combe et al. *Zulu - Active learning from queries competition*. http://labh-curien.univ-st-etienne.fr/zulu/index.php. Version from 01.08.2010 (cit. on p. 153).

[49] Edsger W. Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische Mathematik* 1 (1959), pp. 269–271. DOI: 10.1007/BF01386390 (cit. on p. 72).

[50] Edsger W. Dijkstra. “Chapter I: Notes on Structured Programming”. In: *Structured Programming*. GBR: Academic Press Ltd., 1972, pp. 1–82. ISBN: 0122005503 (cit. on p. 3).

[51] Samuel Drews and Loris D’Antoni. “Learning Symbolic Automata”. In: *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*. Ed. by Axel Legay and Tiziana Margaria. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 173–189. DOI: 10.1007/978-3-662-54577-5_10 (cit. on p. 137).

[52] E. Allen Emerson and Joseph Y. Halpern. “"Sometimes" and "Not Never" Revisited: On Branching Versus Linear Time”. In: *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*. Ed. by John R. Wright et al. ACM Press, 1983, pp. 127–140. DOI: 10.1145/567067.567081 (cit. on p. 133).

[53] “Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)”. In: *Official Journal of the Euro-*

pean Union L119 (2016), pp. 1–88. URL: http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:L:2016:119:TOC (cit. on p. 145).

[54] Yliès Falcone et al. "A Taxonomy for Classifying Runtime Verification Tools". In: *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*. Ed. by Christian Colombo and Martin Leucker. Vol. 11237. Lecture Notes in Computer Science. Springer, 2018, pp. 241–262. DOI: 10.1007/978-3-030-03769-7_14 (cit. on pp. 3, 141).

[55] Lu Feng, Marta Kwiatkowska, and David Parker. "Compositional Verification of Probabilistic Systems Using Learning". In: *Proceedings of the 2010 Seventh International Conference on the Quantitative Evaluation of Systems*. QEST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 133–142. DOI: 10.1109/QEST.2010.24 (cit. on p. 137).

[56] Emmanuel Filiot et al. "Visibly Pushdown Transducers". In: *Journal of Computer and System Sciences* 97 (2018), pp. 147–181. DOI: 10.1016/j.jcss.2018.05.002 (cit. on p. 131).

[57] Paul Fiterău-Broştean and Falk Howar. "Learning-Based Testing the Sliding Window Behavior of TCP Implementations". In: *Critical Systems: Formal Methods and Automated Verification - Joint 22nd International Workshop on Formal Methods for Industrial Critical Systems - and - 17th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2017, Turin, Italy, September 18-20, 2017, Proceedings*. Ed. by Laure Petrucci, Cristina Seceleanu, and Ana Cavalcanti. Vol. 10471. Lecture Notes in Computer Science. Springer, 2017, pp. 185–200. DOI: 10.1007/978-3-319-67113-0_12 (cit. on pp. 4, 137).

[58] Paul Fiterău-Broştean, Ramon Janssen, and Frits W. Vaandrager. "Learning Fragments of the TCP Network Protocol". In: *Formal Methods for Industrial Critical Systems - 19th International Conference, FMICS 2014, Florence, Italy, September 11-12, 2014. Proceedings*. Ed. by Fréd Éric Lang and Francesco Flammini. Vol. 8718. Lecture Notes in Computer Science. Springer, 2014, pp. 78–93. DOI: 10.1007/978-3-319-10702-8_6 (cit. on p. 137).

[59] Markus Frohme and Bernhard Steffen. "A Context-Free Symbiosis of Runtime Verification and Automata Learning". In: Lecture Notes in Computer Science 13065 (2021). Ed. by Ezio Bartocci, Yliès Falcone, and Martin Leucker, pp. 159–181. DOI: 10.1007/978-3-030-87348-6_10 (cit. on pp. 7, 61, 148–151).

[60] Markus Frohme and Bernhard Steffen. "Active Mining of Document Type Definitions". In: *Formal Methods for Industrial Critical Systems - 23rd International Conference, FMICS 2018, Maynooth, Ireland, September 3-4, 2018, Proceedings*. Ed. by Falk Howar and Jiri Barnat. Vol. 11119. Lecture Notes in Computer Science. Springer, 2018, pp. 147–161. DOI: 10.1007/978-3-030-00244-2_10 (cit. on pp. 7, 143–147).

[61] Markus Frohme and Bernhard Steffen. “Compositional learning of mutually recursive procedural systems”. In: *International Journal on Software Tools for Technology Transfer* 23.4 (2021), pp. 521–543. DOI: 10.1007/s10009-021-00634-y (cit. on pp. 5, 27, 30–33, 35–40, 42, 50, 88, 90–93, 95, 98, 100, 155, 158, 160).

[62] Markus Frohme and Bernhard Steffen. “From Languages to Behaviors and Back”. In: Lecture Notes in Computer Science 13560 (2022). Ed. by Nils Jansen, Mariëlle Stoelinga, and Petra van den Bos, pp. 180–200. DOI: 10.1007/978-3-031-15629-8_11 (cit. on pp. 6, 46, 47, 49, 50, 53, 55, 105, 155, 158).

[63] Markus Frohme and Bernhard Steffen. “Never-Stop Context-Free Learning”. In: Lecture Notes in Computer Science 13030 (2021). Ed. by Ernst-Rüdiger Olderog, Bernhard Steffen, and Wang Yi, pp. 164–185. DOI: 10.1007/978-3-030-91384-7_9 (cit. on pp. 6, 148, 151, 152, 155, 158, 160).

[64] Markus Theo Frohme. “Active Automata Learning with Adaptive Distinguishing Sequences”. In: *CoRR* abs/1902.01139 (2019). arXiv: 1902.01139 (cit. on p. 137).

[65] Susumu Fujiwara et al. “Test Selection Based on Finite State Models”. In: *IEEE Transactions on Software Engineering* 17.6 (1991), pp. 591–603. DOI: 10.1109/32.87284 (cit. on p. 135).

[66] Sato Fumiaki et al. “Test Sequence Generation Method for Systems-Based Finite Automata – Single Transition Checking Method Using W Set”. In: *Transactions of the Institute of Electronics, Information and Communication Engineers* J72-B-I.3 (1989), pp. 183–192. DOI: 10.1002/ecja.4410730303 (cit. on p. 135).

[67] Pedro Garcia and Jose Oncina. *Inference of recognizable tree sets*. Tech. rep. Dept. Syst. Inform. Comput., Univ. Politécnica Valencia, Valencia, Spain, DSIC/II/47/93, 1993 (cit. on p. 138).

[68] Angelo Gargantini. “Conformance Testing”. In: Manfred Broy et al. *Model-Based Testing of Reactive Systems:* vol. 3472. Lecture Notes in Computer Science. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005, pp. 87–111. DOI: 10.1007/11498490_5 (cit. on pp. 3, 21, 135, 136).

[69] Marco Gario et al. “Model Checking at Scale: Automated Air Traffic Control Design Space Exploration”. In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 3–22. DOI: 10.1007/978-3-319-41540-6_1 (cit. on p. 3).

[70] Seymour Ginsburg and Michael A. Harrison. “Bracketed Context-Free Languages”. In: *Journal of Computer and System Science* 1.1 (1967), pp. 1–23. DOI: 10.1016/S0022-0000(67)80003-5 (cit. on p. 28).

[71] E. Mark Gold. “Complexity of Automaton Identification from Given Data”. In: *Inf. Control.* 37.3 (1978), pp. 302–320. DOI: 10.1016/S0019-9958(78)90562-4 (cit. on p. 178).

[72] Sheila A. Greibach. “A New Normal-Form Theorem for Context-Free Phrase Structure Grammars”. In: *Journal of the ACM* 12.1 (1965), pp. 42–52. DOI: 10.1145/321250.321254 (cit. on p. 138).

[73] Matthew Hague and C.-H. Luke Ong. “Analysing Mu-Calculus Properties of Pushdown Systems”. In: *Model Checking Software - 17th International SPIN Workshop, Enschede, The Netherlands, September 27-29, 2010. Proceedings*. Ed. by Jaco van de Pol and Michael Weber. Vol. 6349. Lecture Notes in Computer Science. Springer, 2010, pp. 187–192. DOI: 10.1007/978-3-642-16164-3_14 (cit. on p. 135).

[74] Anthony Hall. “Correctness by Construction: Integrating Formality into a Commercial Development Process”. In: *FME 2002: Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark, July 22-24, 2002, Proceedings*. Ed. by Lars-Henrik Eriksson and Peter A. Lindsay. Vol. 2391. Lecture Notes in Computer Science. Springer, 2002, pp. 224–233. DOI: 10.1007/3-540-45614-7_13 (cit. on p. 177).

[75] Anthony Hall and Roderick Chapman. “Correctness by Construction: Developing a Commercial Secure System”. In: *IEEE Softw.* 19.1 (2002), pp. 18–25. DOI: 10.1109/52.976937 (cit. on p. 177).

[76] Matthew Hennessy and Robin Milner. “Algebraic Laws for Nondeterminism and Concurrency”. In: *Journal of the ACM* 32.1 (1985), pp. 137–161. DOI: 10.1145/2455.2460 (cit. on pp. 20, 133).

[77] Marco Henrix. “Performance improvement in automata learning”. Master Thesis. Radboud University, Nijmegen, 2015 (cit. on p. 137).

[78] Thomas A. Henzinger et al. “Temporal-Safety Proofs for Systems Code”. In: *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*. Ed. by Ed Brinksma and Kim Guldstrand Larsen. Vol. 2404. Lecture Notes in Computer Science. Springer, 2002, pp. 526–538. DOI: 10.1007/3-540-45657-0_45 (cit. on p. 3).

[79] Tim Hepher. “Exclusive: A400M probe focuses on impact of accidental data wipe”. In: *Reuters* (June 9, 2015). URL: https://www.reuters.com/article/us-airbus-a400m-idUSKBN0OP2AS20150609 (visited on 2022-12-07) (cit. on p. 1).

[80] Thomas N. Hibbard. “Least Upper Bounds on Minimal Terminal State Experiments for Two Classes of Sequential Machines”. In: *Journal of the ACM* 8.4 (1961), pp. 601–612. DOI: 10.1145/321088.321098 (cit. on p. 136).

[81] Gerard J. Holzmann. “The Model Checker SPIN”. In: *IEEE Trans. Software Eng.* 23.5 (1997), pp. 279–295. DOI: 10.1109/32.588521 (cit. on p. 3).

[82] John Hopcroft. “An N Log N Algorithm for Minimizing States in a Finite Automaton”. In: *Theory of Machines and Computations*. Ed. by Zvi Kohavi and Azaria Paz. Academic Press, 1971, pp. 189–196. DOI: 10.1016/B978-0-12-417750-5.50022-1 (cit. on pp. 16, 30).

[83] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979. ISBN: 0-201-02988-X (cit. on pp. 13, 14, 30).

[84] Falk Howar. "Active Learning of Interface Programs". PhD thesis. TU Dortmund University, 2012. DOI: 10.17877/DE290R-4817 (cit. on pp. 137, 168).

[85] Falk Howar and Bernhard Steffen. "Active Automata Learning as Black-Box Search and Lazy Partition Refinement". In: *A Journey from Process Algebra via Timed Automata to Model Learning - Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*. Ed. by Nils Jansen, Mariëlle Stoelinga, and Petra van den Bos. Vol. 13560. Lecture Notes in Computer Science. Springer, 2022, pp. 321–338. DOI: 10.1007/978-3-031-15629-8_17 (cit. on p. 137).

[86] Falk Howar, Bernhard Steffen, and Maik Merten. "Automata Learning with Automated Alphabet Abstraction Refinement". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ranjit Jhala and David Schmidt. Vol. 6538. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, pp. 263–277. DOI: 10.1007/978-3-642-18275-4_19 (cit. on pp. 119, 127, 137, 176).

[87] Falk Howar, Bernhard Steffen, and Maik Merten. "From ZULU to RERS - Lessons Learned in the ZULU Challenge". In: *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 6415. Lecture Notes in Computer Science. Springer, 2010, pp. 687–704. DOI: 10.1007/978-3-642-16558-0_55 (cit. on p. 153).

[88] Falk Howar et al. "Inferring Canonical Register Automata". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Viktor Koncak and Andrey Rybalchenko. Vol. 7148. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012, pp. 251–266. DOI: 10.1007/978-3-642-27940-9_17 (cit. on p. 137).

[89] Falk Howar et al. "On Handling Data in Automata Learning - Considerations from the CONNECT Perspective". In: *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part II*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 6416. Lecture Notes in Computer Science. Springer, 2010, pp. 221–235. DOI: 10.1007/978-3-642-16561-0_24 (cit. on p. 149).

[90] Falk Howar et al. "The Teachers' Crowd: The Impact of Distributed Oracles on Active Automata Learning". In: *Leveraging Applications of Formal Methods, Verification, and Validation*. Ed. by Reiner Hähnle et al. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2012, pp. 232–247. DOI: 10.1007/978-3-642-34781-8_18 (cit. on pp. 137, 167).

[91] Hardi Hungar, Oliver Niese, and Bernhard Steffen. "Domain-Specific Optimization in Automata Learning". In: *Proc. 15th Int. Conf. on Computer Aided Verification*. Ed. by Warren A. Hunt Jr. and Fabio Somenzi. Vol. 2725. Lecture Notes in Computer Science. Springer Verlag, July 2003, pp. 315–327. DOI: 10.1007/978-3-540-45069-6_31 (cit. on pp. 4, 137).

[92] *ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF*. Tech. rep. 1996. URL: https://www.iso.org/standard/26153.html (cit. on p. 30).

[93] Malte Isberner. "Foundations of Active Automata Learning: An Algorithmic Perspective". PhD thesis. Technical University Dortmund, Germany, 2015. DOI: 10.17877/DE290R-16359 (cit. on pp. 114–117, 124, 131, 137, 138, 156, 160, 162).

[94] Malte Isberner, Falk Howar, and Bernhard Steffen. "Learning register automata: from languages to program structures". In: *Machine Learning* (2013), pp. 1–34. DOI: 10.1007/s10994-013-5419-7 (cit. on p. 137).

[95] Malte Isberner, Falk Howar, and Bernhard Steffen. "The Open-Source LearnLib: A Framework for Active Automata Learning". In: *CAV 2015*. 2015. DOI: 10.1007/978-3-319-21690-4_32 (cit. on pp. 8, 66, 137, 159).

[96] Malte Isberner, Falk Howar, and Bernhard Steffen. "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning". English. In: *Runtime Verification*. Ed. by Borzoo Bonakdarpour and Scott A. Smolka. Vol. 8734. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 307–322. DOI: 10.1007/978-3-319-11164-3_26 (cit. on pp. 6, 124, 137, 148, 151, 168).

[97] Malte Isberner and Bernhard Steffen. "An Abstract Framework for Counterexample Analysis in Active Automata Learning". In: *Proceedings of the 12th International Conference on Grammatical Inference, ICGI 2014, Kyoto, Japan, September 17-19, 2014*. Ed. by Alexander Clark, Makoto Kanazawa, and Ryo Yoshinaka. Vol. 34. JMLR Workshop and Conference Proceedings. JMLR.org, 2014, pp. 79–93. URL: http://proceedings.mlr.press/v34/isberner14a.html (cit. on p. 137).

[98] Hiroki Ishizaka. "Polynomial time Learnability of Simple Deterministic Languages". In: *Machine Learning* 5 (1990), pp. 151–164. DOI: 10.1007/BF00116035 (cit. on p. 138).

[99] Valérie Issarny et al. "CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems". In: *14th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2009, Potsdam, Germany, 2-4 June 2009*. IEEE Computer Society, 2009, pp. 154–161. DOI: 10.1109/ICECCS.2009.44 (cit. on pp. 4, 137, 149).

[100] Gijs Kant et al. “LTSmin: High-Performance Language-Independent Model Checking”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Christel Baier and Cesare Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 692–707. DOI: 10.1007/978-3-662-46681-0_61 (cit. on p. 3).

[101] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. Cambridge, MA, USA: MIT Press, 1994. ISBN: 0-262-11193-4 (cit. on p. 136).

[102] Ali Khalili and Armando Tacchella. “Learning Nondeterministic Mealy Machines”. In: *Proceedings of the 12th International Conference on Grammatical Inference, ICGI 2014, Kyoto, Japan, September 17-19, 2014*. Ed. by Alexander Clark, Makoto Kanazawa, and Ryo Yoshinaka. Vol. 34. JMLR Workshop and Conference Proceedings. JMLR.org, 2014, pp. 109–123. URL: http://proceedings.mlr.press/v34/khalili14a.html (cit. on p. 137).

[103] Florent Kirchner et al. “Frama-C: A software analysis perspective”. In: *Formal Aspects Comput.* 27.3 (2015), pp. 573–609. DOI: 10.1007/s00165-014-0326-7 (cit. on p. 135).

[104] Igor Konstantinovich Klyachko Alexander Anatolevich. Rystsov and M. A. Spivak. “In extremal combinatorial problem associated with the bound on the length of a synchronizing word in an automaton”. In: *Cybernetics and Systems Analysis* 23 (Mar. 1987), pp. 165–171. DOI: 10.1007/BF01071771 (cit. on p. 135).

[105] Savas Konur. “A survey on temporal logics for specifying and verifying real-time systems”. In: *Frontiers of Computer Science* 7.3 (2013), pp. 370–403. DOI: 10.1007/s11704-013-2195-2 (cit. on p. 133).

[106] Philip Koopman. *A Case Study of Toyota Unintended Acceleration and Software Safety*. Nov. 14, 2014. URL: http://chess.eecs.berkeley.edu/pubs/1081.html (visited on 2023-06-19) (cit. on p. 1).

[107] Dexter Kozen. “Results on the Propositional μ-Calculus”. In: *Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proceedings*. Ed. by Mogens Nielsen and Erik Meineche Schmidt. Vol. 140. Lecture Notes in Computer Science. Springer, 1982, pp. 348–359. DOI: 10.1007/BFb0012782 (cit. on pp. 2, 20, 134).

[108] Saul A. Kripke. “Semantical Considerations on Modal Logic”. In: *Acta Philosophica Fennica* 16 (1963), pp. 83–94 (cit. on pp. 2, 19).

[109] Viraj Kumar, Parthasarathy Madhusudan, and Mahesh Viswanathan. “Minimization, Learning, and Conformance Testing of Boolean Programs”. In: *CONCUR 2006 – Concurrency Theory: 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006. Proceedings*. Ed. by Christel Baier and Holger Her-

manns. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 203–217. DOI: 10.1007/11817949_14 (cit. on pp. 134, 137, 138).

[110] Robert P. Kurshan. "Formal Verification in a Commercial Setting". In: *Proceedings of the 34th Annual Design Automation Conference*. DAC '97. Anaheim, California, USA: Association for Computing Machinery, 1997, pp. 258–262. DOI: 10.1145/266021.266089 (cit. on p. 3).

[111] David Lee and Mihalis Yannakakis. "Principles and Methods of Testing Finite State Machines – A Survey". In: *Proceedings of the IEEE* 84.8 (1996), pp. 1090–1123. DOI: 10.1109/5.533956 (cit. on pp. 3, 21, 135).

[112] David Lee and Mihalis Yannakakis. "Testing Finite-State Machines: State Identification and Verification". In: *IEEE Trans. Computers* 43.3 (1994), pp. 306–320. DOI: 10.1109/12.272431 (cit. on p. 137).

[113] Michael Lybecait, Dawid Kopetzki, and Bernhard Steffen. "Design for 'X' through Model Transformation". In: *Proc. of the 8th Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation, Part I Modeling (ISoLA 2018)*. Vol. 11244. Lecture Notes in Computer Science. Springer, 2018, pp. 381–398. DOI: 10.1007/978-3-030-03418-4_23 (cit. on p. 177).

[114] Wolfgang Maass and György Turán. "Lower Bound Methods and Separation Results for On-Line Learning Models". In: *Machine Learning* 9 (1992), pp. 107–145. DOI: 10.1007/BF00992674 (cit. on p. 138).

[115] Oded Maler and Irini-Eleftheria Mens. "Learning Regular Languages over Large Alphabets". In: *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*. Ed. by Erika Ábrahám and Klaus Havelund. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 485–499. DOI: 10.1007/978-3-642-54862-8_41 (cit. on p. 137).

[116] Oded Maler and Amir Pnueli. "On the Learnability of Infinitary Regular Sets". In: *Information and Computation* 118.2 (1995), pp. 316–326. DOI: 10.1006/inco.1995.1070 (cit. on p. 136).

[117] Tiziana Margaria, Harald Raffelt, and Bernhard Steffen. "Knowledge-based relevance filtering for efficient system-level test-based model generation". In: *Innovations in Systems and Software Engineering* 1.2 (2005), pp. 147–156. DOI: 10.1007/s11334-005-0016-y (cit. on p. 137).

[118] Tiziana Margaria et al. "Efficient test-based model generation for legacy reactive systems". In: *HLDVT '04: Proceedings of the High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 95–100. DOI: http://dx.doi.org/10.1109/HLDVT.2004.1431246 (cit. on p. 137).

[119] Nicolas Markey. "Past is for Free: on the Complexity of Verifying Linear Temporal Properties with Past". In: *9th International Workshop on Expressiveness in Concurrency, EXPRESS 2002, Satellite Workshop from CONCUR 2002, Brno, Czech Republic, August 19, 2002*. Ed. by Uwe Nestmann and Prakash Panangaden. Vol. 68. Electronic Notes in Theoretical Computer Science 2. Elsevier, 2002, pp. 87–104. DOI: 10.1016/S1571-0661(05)80366-4 (cit. on p. 133).

[120] Franco Mazzanti, Alessio Ferrari, and Giorgio Oronzo Spagnolo. "Towards formal methods diversity in railways: an experience report with seven frameworks". In: *STTT* 20.3 (2018), pp. 263–288. DOI: 10.1007/s10009-018-0488-3 (cit. on p. 3).

[121] George H. Mealy. "A method for synthesizing sequential circuits". In: *The Bell System Technical Journal* 34.5 (1955), pp. 1045–1079. DOI: 10.1002/j.1538-7305.1955.tb03788.x (cit. on pp. 2, 9, 17).

[122] Jeroen Meijer and Jaco van de Pol. "Sound black-box checking in the LearnLib". In: *Innov. Syst. Softw. Eng.* 15.3-4 (2019), pp. 267–287. DOI: 10.1007/s11334-019-00342-6 (cit. on p. 137).

[123] Karl Meinke, Fei Niu, and Muddassar A. Sindhu. "Learning-Based Software Testing: A Tutorial". In: *Leveraging Applications of Formal Methods, Verification, and Validation - International Workshops, SARS 2011 and MLSC 2011, Held Under the Auspices of ISoLA 2011 in Vienna, Austria, October 17-18, 2011. Revised Selected Papers*. Ed. by Reiner Hähnle et al. Vol. 336. Communications in Computer and Information Science. Springer, 2011, pp. 200–219. DOI: 10.1007/978-3-642-34781-8_16 (cit. on p. 139).

[124] Karl Meinke and Muddassar A. Sindhu. "Incremental Learning-Based Testing for Reactive Systems". In: *Tests and Proofs - 5th International Conference, TAP@TOOLS 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings*. Ed. by Martin Gogolla and Burkhart Wolff. Vol. 6706. Lecture Notes in Computer Science. Springer, 2011, pp. 134–151. DOI: 10.1007/978-3-642-21768-5_11 (cit. on p. 139).

[125] Karl Meinke and Muddassar Azam Sindhu. "LBTest: A Learning-Based Testing Tool for Reactive Systems". In: *IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST), 2013*. Mar. 2013, pp. 447–454. DOI: 10.1109/ICST.2013.62 (cit. on p. 137).

[126] Edward F. Moore. "Gedanken-Experiments on Sequential Machines". In: *Annals of Mathematical Studies* 34 (1956), pp. 129–153 (cit. on pp. 22, 26, 71).

[127] Edi Muskardin et al. "AALpy: An Active Automata Learning Library". In: *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceedings*. Ed. by Zhe Hou and Vijay Ganesh. Vol. 12971. Lecture Notes in Computer Science. Springer, 2021, pp. 67–73. DOI: 10.1007/978-3-030-88885-5_5 (cit. on p. 137).

[128] Muhammad Muzammil Shahbaz. “Reverse Engineering Enhanced State Models of Black Box Software Components to support Integration Testing”. PhD thesis. Institut Polytechnique de Grenoble, Dec. 2008 (cit. on p. 137).

[129] Stefan Naujokat et al. “CINCO: A Simplicity-Driven Approach to Full Generation of Domain-Specific Graphical Modeling Tools”. In: *Software Tools for Technology Transfer* 20.3 (2017), pp. 327–354. DOI: 10.1007/s10009-017-0453-6 (cit. on p. 177).

[130] Anil Nerode. “Linear Automaton Transformations”. In: *Proceedings of the American Mathematical Society* 9.4 (1958), pp. 541–544. DOI: 10.2307/2033204 (cit. on p. 114).

[131] Johannes Neubauer, Stephan Windmüller, and Bernhard Steffen. “Risk-Based Testing via Active Continuous Quality Control”. In: *International Journal on Software Tools for Technology Transfer* 16.5 (2014), pp. 569–591. DOI: 10.1007/s10009-014-0321-6 (cit. on pp. 4, 137).

[132] Rocco De Nicola and Frits W. Vaandrager. “Action versus State based Logics for Transition Systems”. In: *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science, La Roche Posay, France, April 23-27, 1990, Proceedings*. Ed. by Irène Guessarian. Vol. 469. Lecture Notes in Computer Science. Springer, 1990, pp. 407–419. DOI: 10.1007/3-540-53479-2_17 (cit. on pp. 20, 21, 134).

[133] Oliver Niese. “An Integrated Approach to Testing Complex Systems”. PhD thesis. TU Dortmund University, Germany, 2003. DOI: 10.17877/DE290R-14871 (cit. on p. 6).

[134] Robert Paige and Robert Endre Tarjan. “Three Partition Refinement Algorithms”. In: *SIAM J. Comput.* 16.6 (1987), pp. 973–989. DOI: 10.1137/0216062 (cit. on pp. 16, 30).

[135] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. “Black Box Checking”. In: *Formal Methods for Protocol Engineering and Distributed Systems: FORTE XII / PSTV XIX. IFIP Advances in Information and Communication Technology*. Ed. by Jianping Wu, Samuel T. Chanson, and Qiang Gao. Boston, MA: Springer US, 1999, pp. 225–240. DOI: 10.1007/978-0-387-35578-8_13 (cit. on pp. 4, 139).

[136] Andrea Pferscher and Bernhard K. Aichernig. “Fingerprinting Bluetooth Low Energy Devices via Active Automata Learning”. In: *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*. Ed. by Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan. Vol. 13047. Lecture Notes in Computer Science. Springer, 2021, pp. 524–542. DOI: 10.1007/978-3-030-90870-6_28 (cit. on pp. 4, 137).

[137] *Pin - A Dynamic Binary Instrumentation Tool*. URL: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html (visited on 2023-03-19) (cit. on p. 142).

[138] Carl Pixley, Seh-Woong Jeong, and Gary D. Hachtel. "Exact Calculation of Synchronization Sequences Based on Binary Decision Diagrams". In: *Proceedings of the 29th Design Automation Conference, Anaheim, California, USA, June 8-12, 1992*. Ed. by Daniel G. Schweikert. IEEE Computer Society Press, 1992, pp. 620–623. URL: http://portal.acm.org/citation.cfm?id=113938.149645 (cit. on p. 135).

[139] Gordon D. Plotkin. *A Structural Approach to Operational Semantics*. Tech. rep. DAIMI FN-19. University of Aarhus, 1981 (cit. on p. 31).

[140] Amir Pnueli. "The Temporal Logic of Programs". In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. SFCS '77. USA: IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32 (cit. on pp. 2, 133).

[141] Kevin Poulsen. "Software bug contributed to blackout". In: *The Register* (Feb. 12, 2004). URL: https://www.theregister.com/2004/02/12/software_bug_contributed_to_blackout/ (visited on 2022-12-07) (cit. on p. 1).

[142] Dan Quinlan and Chunhua Liao. "The ROSE source-to-source compiler infrastructure". In: *Cetus users and compiler infrastructure workshop, in conjunction with PACT*. 2011 (cit. on p. 3).

[143] Harald Raffelt, Bernhard Steffen, and Tiziana Margaria. "Dynamic Testing Via Automata Learning". In: *Hardware and Software: Verification and Testing*. Ed. by Karen Yorav. Vol. 4899. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pp. 136–152. DOI: 10.1007/978-3-540-77966-7_13 (cit. on pp. 4, 137).

[144] Jean-François Raskin and Frédéric Servais. "Visibly Pushdown Transducers". In: *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*. Ed. by Luca Aceto et al. Vol. 5126. Lecture Notes in Computer Science. Springer, 2008, pp. 386–397. DOI: 10.1007/978-3-540-70583-3_32 (cit. on p. 131).

[145] Arend Rensink. "The GROOVE Simulator: A Tool for State Space Generation". In: *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers*. Ed. by John L. Pfaltz, Manfred Nagl, and Boris Böhlen. Vol. 3062. Lecture Notes in Computer Science. Springer, 2003, pp. 479–485. DOI: 10.1007/978-3-540-25959-6_40 (cit. on p. 156).

[146] Ronald L. Rivest and Robert E. Schapire. "Inference of finite automata using homing sequences". In: *Inf. Comput.* 103.2 (1993), pp. 299–347. DOI: http://dx.doi.org/10.1006/inco.1993.1021 (cit. on pp. 91, 93, 137, 168).

[147] Kristin Yvonne Rozier. “Specification: The Biggest Bottleneck in Formal Methods and Autonomy”. In: *Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers*. Ed. by Sandrine Blazy and Marsha Chechik. Vol. 9971. Lecture Notes in Computer Science. 2016, pp. 8–26. DOI: 10.1007/978-3-319-48869-1_2 (cit. on p. 3).

[148] Igor Konstantinovich Rystsov. “Polynomial Complete Problems in Automata Theory”. In: *Inf. Process. Lett.* 16.3 (1983), pp. 147–151. DOI: 10.1016/0020-0190(83)90067-4 (cit. on p. 136).

[149] Yasubumi Sakakibara. “Efficient Learning of Context-Free Grammars from Positive Structural Examples”. In: *Inf. Comput.* 97.1 (1992), pp. 23–60. DOI: 10.1016/0890-5401(92)90003-X (cit. on p. 138).

[150] Yasubumi Sakakibara. “Learning Context-Free Grammars from Structural Data in Polynomial Time”. In: *Theoretical Computer Science* 76.2-3 (1990), pp. 223–242. DOI: 10.1016/0304-3975(90)90017-C (cit. on p. 138).

[151] Sven Sandberg. “Homing and Synchronizing Sequences”. In: Manfred Broy et al. *Model-Based Testing of Reactive Systems:* vol. 3472. Lecture Notes in Computer Science. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005, pp. 5–33. DOI: 10.1007/11498490_2 (cit. on pp. 135, 136).

[152] Thomas Schlipf et al. “Formal verification made easy”. In: *IBM Journal of Research and Development* 41.4 & 5 (1997), pp. 567–576. DOI: 10.1147/rd.414.0567 (cit. on p. 3).

[153] Thomas Schwentick. “Automata for XML - A survey”. In: *Journal of Computer and System Sciences* 73.3 (2007), pp. 289–315. DOI: 10.1016/j.jcss.2006.10.003 (cit. on p. 179).

[154] Stefan Schwoon. “Model checking pushdown systems”. PhD thesis. Technical University Munich, Germany, 2002 (cit. on p. 3).

[155] Frédéric Servais. “Visibly Pushdown Transducers”. PhD thesis. Université Libre de Bruxelles, Belgium, 2013 (cit. on p. 131).

[156] Muzammil Shahbaz and Roland Groz. “Inferring Mealy Machines”. In: *FM ’09: Proceedings of the 2nd World Congress on Formal Methods*. Eindhoven, The Netherlands: Springer Verlag, 2009, pp. 207–222. DOI: http://dx.doi.org/10.1007/978-3-642-05089-3_14 (cit. on p. 136).

[157] Alberto Rodrigues da Silva. “Model-driven engineering: A survey supported by the unified conceptual model”. In: *Comput. Lang. Syst. Struct.* 43 (2015), pp. 139–155. DOI: 10.1016/j.cl.2015.06.001 (cit. on p. 177).

[158] Wouter Smeenk et al. “Applying Automata Learning to Embedded Control Software”. In: *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*. Ed. by Michael J. Butler, Sylvain Conchon, and Fatiha Zaïdi. Vol. 9407. Lecture Notes in Computer Science. Springer, 2015, pp. 67–83. DOI: 10.1007/978-3-319-25423-4_5 (cit. on pp. 4, 137).

[159] Rick Smetsers et al. “Complementing Model Learning with Mutation-Based Fuzzing”. In: *CoRR* abs/1611.02429 (2016). arXiv: 1611.02429 (cit. on pp. 135, 178).

[160] *SOAP 1.2 Specification by the W3C*. URL: https://www.w3.org/TR/soap12/ (visited on 2023-06-10) (cit. on p. 143).

[161] Fu Song and Tayssir Touili. “PoMMaDe: pushdown model-checking for malware detection”. In: *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. Ed. by Bertrand Meyer, Luciano Baresi, and Mira Mezini. ACM, 2013, pp. 607–610. DOI: 10.1145/2491411.2494599 (cit. on p. 135).

[162] Bernhard Steffen and Alnis Murtovi. “M3C: Modal Meta Model Checking”. In: *Formal Methods for Industrial Critical Systems - 23rd International Conference, FMICS 2018, Maynooth, Ireland, September 3-4, 2018, Proceedings*. Ed. by Falk Howar and Jiri Barnat. Vol. 11119. Lecture Notes in Computer Science. Springer, 2018, pp. 223–241. DOI: 10.1007/978-3-030-00244-2_15 (cit. on pp. 3, 8, 63, 66, 67, 135).

[163] Dejvuth Suwimonteerabuth, Javier Esparza, and Stefan Schwoon. “Symbolic Context-Bounded Analysis of Multithreaded Java Programs”. In: *Model Checking Software, 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008, Proceedings*. Ed. by Klaus Havelund, Rupak Majumdar, and Jens Palsberg. Vol. 5156. Lecture Notes in Computer Science. Springer, 2008, pp. 270–287. DOI: 10.1007/978-3-540-85114-1_19 (cit. on p. 135).

[164] Martin Tappler et al. “L^*-Based Learning of Markov Decision Processes”. In: *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*. Ed. by Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira. Vol. 11800. Lecture Notes in Computer Science. Springer, 2019, pp. 651–669. DOI: 10.1007/978-3-030-30942-8_38 (cit. on p. 137).

[165] Martin Tappler et al. “Active Model Learning of Stochastic Reactive Systems”. In: *Software Engineering and Formal Methods - 19th International Conference, SEFM 2021, Virtual Event, December 6-10, 2021, Proceedings*. Ed. by Radu Calinescu and Corina S. Pasareanu. Vol. 13085. Lecture Notes in Computer Science. Springer, 2021, pp. 481–500. DOI: 10.1007/978-3-030-92124-8_27 (cit. on p. 137).

[166] Tim Tegeler et al. “Product Line Verification via Modal Meta Model Checking”. In: *From Software Engineering to Formal Methods and Tools, and Back: Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday*. Ed. by Maurice H. ter Beek, Alessandro Fantechi, and Laura Semini. Cham: Springer International Publishing, 2019, pp. 313–337. DOI: 10.1007/978-3-030-30985-5_19 (cit. on pp. 145, 147).

[167] Max Tijssen. “Automatic modeling of SSH implementations with state machine learning algorithms”. Bachelor Thesis. Radboud University, Nijmegen, 2014 (cit. on pp. 4, 137).

[168] *Valgrind*. URL: https://valgrind.org/ (visited on 2023-03-19) (cit. on p. 142).

[169] Abhay Vardhan and Mahesh Viswanathan. “LEVER: A Tool for Learning Based Verification”. In: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Ed. by Thomas Ball and Robert B. Jones. Vol. 4144. Lecture Notes in Computer Science. Springer, 2006, pp. 471–474. DOI: 10.1007/11817963_43 (cit. on p. 137).

[170] Moshe Y. Vardi. “Branching vs. Linear Time: Final Showdown”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*. Ed. by Tiziana Margaria and Wang Yi. Vol. 2031. Lecture Notes in Computer Science. Springer, 2001, pp. 1–22. DOI: 10.1007/3-540-45319-9_1 (cit. on pp. 3, 133, 155).

[171] Juan Miguel Vilar. “Query learning of subsequential transducers”. In: *Grammatical Inference: Learning Syntax from Sentences, 3rd International Colloquium, ICGI-96, Montpellier, France, September 25-27, 1996, Proceedings*. Ed. by Laurent Miclet and Colin de la Higuera. Vol. 1147. Lecture Notes in Computer Science. Springer, 1996, pp. 72–83. DOI: 10.1007/BFb0033343 (cit. on p. 175).

[172] Stephan Windmüller et al. “Active Continuous Quality Control”. In: *16th International ACM SIGSOFT Symposium on Component-Based Software Engineering*. CBSE ’13. New York, NY, USA: ACM SIGSOFT, 2013, pp. 111–120. DOI: 10.1145/2465449.2465469 (cit. on pp. 4, 137).

[173] Niklaus Wirth. “What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?” In: *Commun. ACM* 20.11 (1977), pp. 822–823. DOI: 10.1145/359863.359883 (cit. on p. 30).

[174] Andreas Zeller. “Specifications for Free”. In: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. Ed. by Mihaela Gheorghiu Bobaru et al. Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 2–12. DOI: 10.1007/978-3-642-20398-5_2 (cit. on p. 3).

[175] Yang Zhao and Kristin Yvonne Rozier. “Formal Specification and Verification of a Coordination Protocol for an Automated Air Traffic Control System”. In: *ECEASST* 53 (2012). DOI: 10.14279/tuj.eceasst.53.787 (cit. on p. 3).