Learning Finite State Machine Specifications from Test Cases

Dissertation

zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultäten
der Georg-August-Universität zu Göttingen

vorgelegt von

Edith Benedicta Maria Werner
aus Nürnberg

Göttingen
im Mai 2010

Referent: Prof. Dr. Jens Grabowski,
Universität Göttingen.

Korreferent: Prof. Dr. Stephan Waack,
Universität Göttingen.

Tag der mündlichen Prüfung: 1. Juni 2010

Abstract

Up-to-date software systems are often modular and need to be changeable. While modularity is believed to reduce software production costs, it also leads to increased difficulty in testing, as the future uses of a module are getting more varied and therefore harder to guess. Also, reused modules are often represented as black boxes, whose interior structure is hidden from the system. In consequence, on reusing a module, the testing focuses on integration testing and monitoring the interfaces of the module.

In order to monitor a system, a model of the system is needed to compare the observed traces to. However, with the advent of agile development methods and decreasing time to market, formal models of a system are seldom available. While formal modeling has not been widely adopted in industry, the importance of testing has increased, in part thanks to the same agile development methods that obsolete explicit modeling. An example is the test first paradigm of eXtreme Programming, which requires that the tests for any software have to be written before the software itself. Therefore, test cases are available even for systems without a formal model.

In consequence, we propose to generate a system model suitable for monitoring from a test suite for the system. The approach is based on automata learning. Angluin's learning algorithm is used to generate an appropriate model, while state-merging methods are applied to represent the test cases in a format that can be processed by the learning algorithm.

Both Angluin's algorithm and state-merging are tailored to the characteristics of testing. For Angluin's algorithm, this comprises a mapping of the query mechanisms onto a test suite. The state-merging is used to construct a generic representation of arbitrary test suites by exploiting the properties of a given test specification language for a better coverage. The approach is implemented in a prototypical tool and validated by a case study.

Zusammenfassung

Moderne Software-Systeme sind häufig modular aufgebaut, müssen dabei aber Ansprüchen an Wartbarkeit und Änderbarkeit genügen. Während man einerseits annimmt, dass modulare Software mit geringerem Kostenaufwand hergestellt werden kann, wird andererseits die Komplexität des Testens erhöht, da die zukünftigen Anwendungen eines Moduls sich nur schwer vorhersagen lassen. Dazu kommt, dass Module häufig als abgeschlossene Bausteine wiederverwendet werden, so dass die innere Struktur der Module dem System verborgen bleibt. Daher konzentriert sich der Test eines wiederverwendeten Moduls oft auf den Integrationstest und das Beobachten der Schnittstellen des Moduls (*Monitoring*).

Um das System-Verhalten an der beobachteten Schnittstelle bewerten zu können, wird ein formales Modell des Systems benötigt, um die beobachteten Ereignisfolgen damit zu vergleichen. Leider sind formale Modelle nur selten verfügbar, da durch die Anwendung agiler Entwicklungsmethoden und die immer kürzer werdenden Entwicklungszeiten häufig keine formalen Modelle erstellt werden. Gleichzeitig hat sich das Testen von Software immer mehr durchgesetzt, so dass für die meisten Software-Systeme eine Sammlung von Testfällen vorliegt.

Die vorliegende Arbeit schlägt eine Methode vor, mit deren Hilfe aus den Testfällen eines Software-Systems ein formales Modell errechnet werden kann. Die Methode basiert auf Ansätzen zum maschinellen Lernen von endlichen Automaten. Ein Lernalgorithmus, der zuerst von Dana Angluin vorgeschlagen wurde, erzeugt das formale Modell, während Methoden zur Zustands-Vereinigung die Testfälle in eine für den Lernalgorithmus geeignete Datenstruktur umwandeln.

Sowohl der Lernalgorithmus als auch die Methoden zur Zustands-Vereinigung werden an die Eigenschaften des Testens angepasst. Für den Lernalgorithmus von Dana Angluin bedeutet das, dass die Fragemechanismen auf die Testfälle abgebildet werden müssen. Die Zustands-Vereinigung wird benutzt um eine generische Repräsentation beliebiger Testfälle zu errechnen, wobei die semantischen Eigenschaften der Testsprache ausgenutzt werden um eine bessere Überdeckung des Zielmodells zu erhalten. Der kombinierte Ansatz wird in einem prototypischen Werkzeug implementiert und durch eine Fallstudie belegt.

Acknowledgments

Like Alice in Wonderland, the research for this thesis has taken me to various unknown shores and topics. Many friends and colleagues have accompanied me along the path of my scientific journey, and to all of them I offer my thanks for their company.

First and foremost, I want to give my thanks to my supervisor, Prof. Dr. Jens Grabowski. Without his patience and persistence, this thesis would never have been completed.

I am also very grateful to all colleagues who offered their advice on this thesis. Franz Schenk was the first one to read my initial efforts, and his comments on structure have greatly eased my task. Steffen Herbold proved to be the person with the most insight on the mathematical foundations of my research topic; therefore he was the one to spot any theoretical inconsistencies. Philip Makedonski has helped to polish my English phrasing. Special thanks are due to Prof. Dr. Wolfgang May for his valuable comments on correctly formulating mathematical definitions.

Many grateful thoughts also go to all my friends and family, whose constant support and never ending inquiries as to the progress of my thesis helped me finishing it.

Contents

List of Figures

List of Tables

List of Definitions

List of Algorithms

List of Acronyms

CPE	conference protocol entity
CSP	Constraint Satisfaction Problem
DFA	deterministic finite automaton
EFSM	extended finite state machine
FA	finite automaton
FREE	flattened regular expression
FSM	finite state machine
IDE	integrated development environment
LTS	labeled transition system
MAT	minimally adequate teacher
MSC	Message Sequence Chart
NFA	nondeterministic finite automaton
PDU	protocol data unit
SUT	system under test
TTCN-3	Testing and Test Control Notation
UML	Unified Modeling Language

1 Introduction

Today, software systems are generally designed to be modular and reusable. Modularity is used to divide large systems into manageable portions, based on the assumption that cohesive modules are easier to specify, easier to implement and easier to maintain. At the same time, specifying self-contained modules almost directly leads to reuse of modules, as modules can then be regarded as building blocks that can be put together as needed. The ultimate scenario of a modular, reusable system is a web service, where simple services are accessed as needed by various clients and orchestrated into larger systems that can change at any moment.

While this vision of ultimate flexibility is clearly attractive, there are also drawbacks. Where the intended scope and responsibility of a software system before was mostly clearly confined and foreseeable, the further usage of a module is difficult to anticipate. Also, rare events that may lead to failures often cannot be tested beforehand, as the time and resources used in testing are limited or the conditions that lead to the rare event cannot be reproduced under laboratory conditions. For all these reasons, it may be advisable to monitor a system for some time after its deployment.

A monitor for a system needs an oracle that accepts or rejects the observed behavior, e.g. a system model that accepts or rejects the observed traces of the monitored system. Unfortunately, the same dynamic software development processes leading to dynamic modular systems also minimize the generation of formal models, as the specification of a formal model needs both time and expertise. Generating a formal model in retrospect for an already running system is even harder, as the real implementation often deviates from the original specification.

A promising approach for the reconstruction of system models is to use learning algorithms, as has been shown, for example, in [CM96], [HNS03], and [SLG07]. However, all those approaches use learning algorithms to generate test cases for the active probing of the system under test. This has two major drawbacks. First, active probing of a running system may interfere with the system's normal operation and thereby cause erratic behavior or even breakdowns. Also, the learned model will reflect the real implementation, and is therefore not apt to be used as a means to judge the behavior of the system.

In contrast, this thesis proposes a method for learning a system model from the system's test cases without probing the system under test itself. Test cases are almost always available and often more consistent to the system than any other model. Also, they usually take into account all of the system's possible reactions to a stimulus, thereby classifying the anticipated correct reactions as accepted behavior and the incorrect or unexpected reactions as rejected behavior. Simply put, a system model for passive testing is generated from the artifacts used in active testing.

1.1 Contribution of this Thesis

This thesis defines a learning approach to the reconstruction of a system model from the system's test cases. We propose a hybrid approach, which is driven by Angluin's learning algorithm [Ang87] but uses state-merging to enhance the sample space. The contribution comprises:

- An adaptation of Angluin's learning algorithm to the domain of testing, which comprises the redefinition of Angluin's query mechanisms for finite and infinite, i.e. cyclic, test case behavior.
- A collection of state-merging techniques, inspired by Biermann's method [BK76], termed *semantic state-merging*, which exploit the semantic properties of test cases to enlarge the sample space. As an interface between Angluin's learning algorithm and the state-merging techniques, a generic data structure is introduced, the *trace graph*, which is based on the prefix tree acceptor used in state-merging algorithms and tailored to optimally represent the test cases used in the learning process.
- A prototypical implementation and a case study.

This hybrid approach combines the advantages of both its sources. While the adaptation of Angluin's learning algorithm is generic to the domain of testing and can be applied to any test specification language, it is in consequence also restricted to the small common subset of the expressiveness of different test languages. In contrast, semantic state-merging can be specifically tailored to a given test specification language and therefore exploit the specific semantic properties of this test language in addition to the structure of the test cases, but in consequence the generated model is also depending on the test language. In addition, Angluin's algorithm always generates a minimal automaton.

Therefore, Angluin's algorithm is used as a front-end, efficiently generating a generic model, and the state-merging is used as a back-end, mining the test cases. The trace graph data structure serves as a bridge between the two approaches.

1.2 Impact

Scientific papers on the results of this thesis have been peer-reviewed and published on international conferences and workshops. In the following, we list the papers with venue, title, reference, topic, and relate the topic of the paper to the contents of this thesis.

- SV04: *Self-adaptive Functional Testing of Services Working in Continuously Changing Contexts.* Werner, Neukirchen, and Grabowski (2004) [WNG04].
  The paper introduces the idea of using test cases to reconstruct a system model for system monitoring.

- MOTES 08: *Using Learning Techniques to Generate System Models for Online Testing.* Werner, Polonski, and Grabowski (2008) [WPG08].
  In this paper, the approach to learning from test cases is introduced and the first adaptation of Angluin's algorithm is presented. A first implementation and proof-of-concept example are also included. Section 4.2 is based on the contents of this paper.

In the context of this thesis, several student projects were initiated and supervised by the author:

- Sergei Polonski [Pol08] formulated the first adaption of Angluin's learning algorithm in his Master's thesis and implemented it for Boolean signals. He defined the estimation formula for the needed minimal length of the test case traces and identified the relevance of cycles. The results of his thesis form the basis of Section 4.2 and led to the explicit integration of infinite traces (Section 4.3).

- Sanaz Karimkhani, Christian Otto, Sven Withus, and Hang Zhang in their students project [AOWZ09] upgraded the basic implementation to accommodate arbitrary signals in the test cases. They also defined the foundations for the trace tree data structure to represent test cases as described in Section 5.1.1.

- Christian Otto, in his bachelor thesis [Ott09], extended the trace tree structure to include cycles (Section 5.2.4) and implemented the first adjustments for the test language *Testing and Test Control Notation (TTCN-3).*

1.3 Structure of the Thesis

The remainder of this thesis is structured as follows. Chapter 2 gives an overview of the state of the art in automata learning. The foundations of the hybrid learning approach are described in Chapter 3. We provide definitions for the basic theoretical concepts and an overview on the background of our work, the areas of software engineering and testing. Additionally, we give a preliminary introduction to machine learning and explain in detail the two learning algorithms that are adapted for learning from test cases.

The adaptation of Angluin's learning algorithm to learning from test cases is presented in detail in Chapter 4. In the beginning of the chapter, the properties of the learning algorithm are analyzed to identify the necessary adaptations and the properties of test cases are analyzed with respect to their influence on the learning procedure. Based on this analysis, the query mechanisms of Angluin's learning algorithm are redefined for learning from test cases. As a first optimization, cycles in the test cases are exploited for the learning process.

Chapter 5 focuses on the representation of test cases for the learning process. We introduce a data structure that allows the compact storage of our testing data while at the same time simplifying the learning process. The relations of test case properties to the data structure are analyzed and used to preprocess the data structure, thereby enlarging the sample space.

Subsequently, in Chapter 6, a prototypical implementation is presented. Based on an academic example system, the conference protocol, the properties of the hybrid learning approach are validated.

Chapter 7 provides a discussion of our results and identifies constraints that emerged during the experimentation. We also describe where our work can profit from other existing research.

Finally, in Chapter 8, we conclude this thesis with a summary and an outlook on future work.

2 State of the Art

The reconstruction of a deterministic finite automaton from given data has applications in different areas. Most of the basic research stems from the field of language inference, where a formal language is to be identified from positive and negative examples of the language. In recent years, those algorithms have been adapted for reverse-engineering, process analysis, and other applications. Learning algorithms can be classified according to three main features.

- *The resulting model:* Besides deterministic finite automata, there are algorithms for learning formal grammars, trees, and various types of logic formula.
- *The learning procedure:* Two basic approaches can be distinguished. *Synthesis* is based on identifying and merging similar states. *Induction*[1] uses counter-examples to refine the generated automaton by splitting states.
- *The input data:* The main difference is whether an algorithm uses positive examples only, positive and negative examples, or statistical information. Other than that, a variety of differently structured data is used.

As deterministic finite automata are among the most expressive models generated by learning, optimizations based on the restrictions of other target models cannot be transferred. Therefore, this overview on the related work focuses on approaches to learning automata.

The nomenclature on learning procedures is by no means uniform, but depends on the scientific community referencing the procedure. In language theory, the term “inference” is used for both state-merging and state-splitting techniques. The same holds for the terms “learning” and “construction”, which are used inconsistently, depending on the aspect of the approach the respective authors want to emphasize.

Since Gold introduced the concept of “identification in the limit” [Gol67], approaches to learning automata or other classifications have found a wide reception. Therefore, a

[1] In machine learning, “induction” refers to the induction of states, not to mathematical induction.

number of surveys on the topic are available, giving an introduction into the field [AS83, BDGW97, dlH05, Leu07, Pit89].

2.1 Synthesis - State Merging Approaches

The notion of automata synthesis is usually referred to Biermann et al. [BBP75, BK76, PB76]. The basic idea of this approach is to analyze examples of the target automaton, identify possible states, and merge similar states until the remaining states are considered to be sufficiently distinct. Since then, his approach has been widely discussed and adapted to various settings.

Bauer [Bau85] defines an algorithm for the synthesis of a procedure from example computations. He concentrates on soundness and completeness of his algorithm, but also mentions that the well-formedness of the input traces influences the solution of the synthesis problems of loop detection, renaming, and superfluous assignments.

Koskimies and Mäkinen [KM94] construct state machines from scenario trace diagrams using positive data only. They note themselves that distinguishing between desired and undesired merges is very hard without explicit information in the traces. Their trace diagrams consist of interleaved actions and events, but only linear sequences are considered.

Carrasco and Oncina [CO96] propose the algorithm “rlips” for inference from stochastic samples. They build a prefix tree automaton, but can also identify cycles.

Alur, Etessami, and Yannakakis [AEY00] synthesize automata from Message Sequence Charts [ITU99] to check for completeness and realizability of scenario-based specifications. The Message Sequence Charts are linearized and then merged into concurrent automata. Their focus is on providing feedback to software engineers for the specification of scenarios and in automating the generation of state machines from scenarios.

Krüger et al. [FK01, KGSB98, KM03] use Message Sequence Charts to synthesize state charts. The Message Sequence Charts are extended by guards, which are then mapped onto states of an automaton. Their goal is an enhancement of the software specification process, by providing techniques to check for consistency between Message Sequence Charts and to find behavior implied by the specified Message Sequence Charts but not explicitly specified. Along similar lines is the work of Uchitel, Brunet, and Chechik [UBC07], who generate modal transition systems, a variation of labeled transition systems, from Message Sequence Charts.

Coste and Fredouille [CF03] generate unambiguous automata, an intermediate class of automata between deterministic finite automata and nondeterministic finite automata. The advantage of nondeterministic finite automata is a more compact representation and they are also better suited for some application domains. The authors also research the usage of domain and typing information for the enhancement of the learning process [CFKdlH04, KdlH02].

In [DKU06], Duarte, Kramer, and Uchitel extract a labeled transition system model automatically from control flow information and traces by identifying contexts which are defined as a combination of a point in the control flow and a set of attributes constituting a system state. The traces are generated by executing tests on the software and capturing the system trace via code instrumentation.

Delamare, Baudry, and Traon [DBT06] use trace analysis to construct UML 2.0 sequence diagrams from execution traces. To identify the states of the software, they use state vectors that consist of characterizing attributes. A tracing tool is applied to catch the systems traces while a human user inputs data to the software. However, no rule for the generation of inputs is given.

Hallal, Boroday, Petrenko, and Ulrich [HBPU06] use traces captured during test execution for checking properties. The event traces are constructed as a system of communicating automata by computing a partial ordering on the events. An industrial case study using this technique is published in [UP07].

Garcia et al. [GdPAR08] present a state-merging algorithm that generates a nondeterministic finite automaton. Their algorithm is independent of the order of state-merging, provided that a universal sample is available that identifies the state partitions.

2.2 Process Mining

In the area of process mining, a similar approach to state-merging is used. The main difference is that the data for process mining approaches is usually collected by monitoring the processes in question; therefore only positive examples are available.

Cook et al. [CDLW04, CW95, CW98] use state-merging techniques for their process mining, but the states to be merged are selected according to statistical metrics on the data sample.

Van der Aalst et al. [vdA09, vdAWM04] use event analysis of system traces for the generation of models. They develop partial orderings among the different events and use a technique similar to the state-merging of Biermann.

2.3 Induction - State Splitting Approaches

Many approaches in automata inference go back to Angluin [Ang87]. The main idea is an incremental discovery of the concept by asking questions to a teacher. In the following, an overview on the most interesting refinements and applications to Angluin's algorithm is given.

Rivest and Schapire [RS93] adapt Angluin's approach for systems without a reset by using homing sequences. Their example for a system without reset is a robot exploring unknown terrain. In [RS94], the authors adapt their approach to automata where states are represented as vectors of observable binary local state variables. They also introduce another concept of equivalence, *test equivalence*, to suit their representation, where two states belong to the same equivalence class if testing the local state variables yields identical results for both states.

Intelligent agents that generate a model of their opponent's behavior by learning from the so far observed behavior are proposed by Carmel and Markovich [CM96]. Based on Angluin's algorithm, an unsupervised learning algorithm is developed. The accumulative algorithm is extended such that input sequences contradicting the so far learned behavior can be integrated into the model without having to regenerate it.

Freund et al. [FKR$^+$97] generalize the learning algorithm for random walks. In their model, the learner is not allowed to experiment with the machine, but has to use observed system traces. They propose two algorithms, one with and the other without the possibility to reset the target machine to a fixed state. The algorithm without reset uses a local homing sequence, which identifies the finite state of a sequence if a certain output pattern is matched.

Alur, Madhusudan, and Nam [AMN05, NMA08] address the verification of component-based system using assumption-guarantee reasoning. Assumptions are learned by a combination of Angluin's learning algorithm and symbolic model checking, where membership queries and equivalence queries are answered by a model checker.

Sinha and Clarke [SC07] also apply Angluin's learning algorithm to verification. Their main focus is on ameliorating the alphabet explosion problem, which occurs in interacting systems when the learner has to cope with many shared communication variables. The proposed solution is *lazy learning*, where alphabet symbols are clustered and explored symbolically and partitioned in the same fashion as the states in Angluin's original algorithm. Recent work by Chen et al. [CFC$^+$09] amplifies these results by identifying minimal separating deterministic finite automata, and therefore minimal assumptions for compositional verification.

Grinchtein, Leucker, and Piterman [GLP06] use learning to identify network invariants satisfying a given safety property. They overcome the problem of ambiguous information by combining both Angluin's and Biermann's methods, the former to gather information by querying, the latter to infer a more compact representation from possibly ambiguous data gathered.

Bollig et al. [BKKL07, BKKL08] use learning to infer a message-passing automaton from Message Sequence Charts. Their focus is on supporting the software specification process. By entering Message Sequence Charts into an interactive user interface, the user can successively specify a software system. Recent work of Bollig et al. [BHKL09] proposes an adaptation of Angluin's algorithm to the learning of nondeterministic finite automata.

2.4 Machine Learning Approaches in Testing

For many testing techniques, a model of the system under test is a mandatory source. However, models often are not available or out of date due to last minute changes to the implementation. The idea of reconstructing a model from any available source is therefore not new. A common technique is for example the reverse-engineering of models from the source code. For systems whose source code is not available, some approaches based on machine learning have been proposed. A common property of all of them is that the system under test itself is used as the teaching oracle, therefore the generated models are dependent on the implementation.

Hagerer, Hungar, Niese, and Steffen [HHNS02] propose a technique based on regular extrapolation. They aim at reconstructing a model for an older version of the system under test and then to use the model to generate test cases for regression testing a newer version of the system under test. They mine observed system traces and generate a model by abstracting from details. The resulting model is checked for consistency against expert specifications given in linear-time temporal logic and then validated by generating test cases from the model that are executed on the system under test. In a later publication [HNS03], the authors show that automata learning can be successfully optimized to domain-specific structures.

Griffeth, Cantor, and Djouvas [GCD06] use learning techniques to generate an automaton to use in network testing. They define some a priori requirements for the network, which they state as trace properties. Then, they generate tests according to the properties and observe the network's reactions. From the observed network traces, a model is learned, which is then used to predict the further behavior of the network and to generate new test cases. In each iteration, the model is refined from the reactions to previously generated test cases, and then the refined model is used to generate new test cases.

Berg, Jonsson, and Raffelt [BJR06, BJR08] refine Angluin's Algorithm for automata with parameters, where the parameter space is unbound, thus yielding possible infinite state spaces. They propose a two-step approach. In the first step, a finite state Mealy automaton is inferred from a small subset of the parameter domain using Angluin's approach. Then, in the second step, the finite state Mealy automaton is transferred into a more compact symbolic presentation with infinite state.

Shahbaz, Li, and Groz [LGS06, SLG07] present a method to infer parameterized models based on Angluin's algorithm. Focusing on integration testing, they construct models for each of the system's components and then merge the models. Membership queries are realized by testing the system under test itself. In [GLPS08] the authors integrate their inference method into a framework for modular system verification.

2.5 Classification of the Contribution of this Thesis

The aim of the learning methodology presented in this thesis is to reconstruct a model of a system from the test cases used to test the system. The similarity between using a model to generate test cases and learning a deterministic finite automaton from traces seems likely. Therefore, Angluin's algorithm in particular has been adapted to the domain of testing before [BJR06, BJR08, GLPS08, LGS06, SLG07]. However, in all those approaches, the researchers use the learning algorithm to generate test cases that are subsequently executed against the system under test whose model is to be discovered, so that the system under test itself is the oracle for the acceptability of a given behavior. While this is suitable for discovering a system model for integration testing, we want to establish a model to be used in online monitoring. To this end, we need a model that is independent from the implementation itself. Therefore, we use a test suite that was developed due to external criteria as input to the adapted learning algorithm.

As this somewhat contradicts the basic idea of an online learning approach, which is to compensate potentially incomplete data by generating queries on its own, the idea of state-merging is adapted to enhance the sample space. In contrast to the classical state-merging approaches, which merge states based on the structural information contained in the sample, we exploit the semantic properties of the data.

3 Foundations

The idea of learning models from test cases connects a number of different research areas. The notion of automata inference was developed by computer language specialists, at first with the idea of recognizing a language from a set of example phrases. Definitions and research regarding models are maintained mostly in the software engineering community, which overlaps in part with the software test community where testing and test methods are to be found. The theoretical background of automata, formal languages, and algorithms is used by all researchers, although we observe that definitions vary slightly.

The main problem of a research topic at the intersection of established research areas is that the established vocabulary and keywords can overlap. Therefore, the intention of this chapter is threefold. Section 3.1 provides the theoretical definitions that form the foundation of our own work. In Section 3.2, we give an overview on software engineering methods. Subsequently, Section3.3 defines the notion of traces with respect to automata and to software systems. An overview on software testing is presented in Section 3.4. The areas of software engineering and software testing form the background of our adaptation of the learning algorithm. We introduce the established vocabulary of these domains and describe some basic ideas that we draw upon in our adaptation. Section 3.5 introduces the fundamental terms and definitions in machine learning. Lastly, in Sections 3.6 and 3.7, we present the two learning approaches on which our own work is based. As especially the properties of Angluin's algorithm, which is described in Section 3.7, are essential to our adaptation, this method is described in some detail.

3.1 Preliminary Definitions and Nomenclature

There are some terms in computer science that are widely used and therefore assumed to be known. However, the details of the definitions vary with different publications. Within this thesis, we will adhere to the following definitions of languages, graphs, trees, automata, and traces [DR95, HMU06].

3.1.1 Words and Languages

Definition 1 (Word) Given a finite set of symbols Σ, also called *alphabet*, a *word* is any finite sequence of symbols $a \in \Sigma$. The empty word is denoted by ϵ. For any word w, $|w|$ denotes the length of w and $w[i]$, $i \in \mathbb{N}$ and $1 \leq i \leq |w|$ denotes the symbol at the ith position in w. The first symbol in w is $w[1]$ and the last symbol is $w[|w|]$. □

As a finite sequence of letters is known as *string* in most programming languages, this term is also often used to refer to words.

Definition 2 (Formal Language) The set of all possible words over a given alphabet Σ, including the empty word, is denoted as Σ^*. A *formal language* L over Σ is an arbitrary subset of Σ^*. □

Definition 3 (Concatenation of Words) The *concatenation* of two arbitrary words w_1, w_2, denoted as $w_1 \oplus w_2$, is a word w where

$$w[i] = \begin{cases} w_1[i], & \text{if } 1 \leq i \leq |w_1| \\ w_2[i - |w_1|], & \text{if } |w_1| \leq i \leq |w| \,. \end{cases}$$

The length of the concatenated word w is the sum of the length of its parts, $|w| = |w_1| + |w_2|$. □

Definition 4 (Projection) Let Σ be an alphabet and w a word over an arbitrary alphabet. The *projection* $\pi_\Sigma(w)$ of the word w onto the alphabet Σ is defined as follows:

$$\pi_\Sigma(w) = \begin{cases} \epsilon, & \text{if } w = \epsilon, \\ \pi_\Sigma(u), & \text{if } w = u \oplus a, a \notin \Sigma \,, \\ \pi_\Sigma(u) \oplus a, & \text{if } w = u \oplus a, a \in \Sigma \,. \end{cases}$$

Informally described, a projection of a word w onto an alphabet Σ deletes from w all symbols that are not in Σ. □

Definition 5 (Prefix) A word w_p is called a *prefix* of another word w, if there is a third word w' so that $w_p \oplus w' = w$. For any language L over Σ, the set of prefixes is defined as

$$\Pr(L) = \{w \in \Sigma^* \mid \exists w' \in \Sigma^*, w \oplus w' \in L\} \,.$$ □

Definition 6 (Suffix) A word w_s is called a *suffix* of another word w, if there is a third word w' so that $w' \oplus w_s = w$. For any language L over Σ, the set of suffixes is defined as

$$\mathrm{Suf}(L) = \{w \in \Sigma^* \mid \exists w' \in \Sigma^*, w' \oplus w \in L\}. \qquad \square$$

3.1.2 Graphs

Graphs are used to represent various concepts in computer science: networks, computer systems, computer programs, and many more. They also provide a means to define some common notions in a concise way.

Definition 7 (Directed Graph) A *directed graph* $\mathcal{G}$, also called a *digraph*, is a tuple (N, E), where N is a set of *nodes* and $E \subseteq N \times N$ is a set of ordered pairs of nodes called *edges*. An edge $e = (n_1, n_2)$, also denoted by $n_1 \to n_2$, is considered to be directed from n_1 to n_2. We call n_1 the source and n_2 the target of edge e. □

Both nodes and edges of a graph can be labeled with certain information. Suitable labeling functions will be defined as needed. The notation $n_1 \xrightarrow{l} n_2$ denotes an edge from node n_1 to node n_2, which is labeled with l.

Definition 8 (Path) Given a graph $\mathcal{G} = (N, E)$, a *path* is a sequence of nodes $n_1, n_2, ..., n_k$, $n_i \in N$ and $k \in \mathbb{N}$, where for every pair of subsequent nodes (n_i, n_{i+1}) in the sequence there is an edge $e \in E$ in the graph, $e = (n_i, n_{i+1})$. The *length* of a path is the number of nodes it contains, $|n_1, n_2, ..., n_k| = k$. □

Definition 9 (Simple Cycle) Given a graph $\mathcal{G} = (N, E)$, a path $(n_0, n_1, ..., n_k)$, $n_i \in N$ and $k \in \mathbb{N}$, is called a *simple cycle*, when $n_0 = n_k$ and all nodes n_i where $i < k$ are mutually distinct. □

Definition 10 (Predecessor, Successor) Given a graph $\mathcal{G} = (N, E)$, a node $n_1 \in N$ is called *direct predecessor* of another node $n_2 \in N$, if there is an edge directed from n_1 to n_2, $\exists e \in E : e = (n_1, n_2)$, or *predecessor*, if there is a path of arbitrary length from n_1 to n_2. The terms *successor* and *direct successor* are defined accordingly. □

3.1.3 Trees

A graph theory, a tree is viewed as a special case of a graph and the definitions are formulated accordingly. For programming purposes, trees are usually defined as recursive data structures. The definitions are compatible to each other, but focus on different properties of the tree. In this thesis, the following definitions for trees are used.

Definition 11 (Tree) A *tree* $\mathcal{T}$ is a directed graph $\mathcal{G} = (N, E)$ with the following properties [HMU06].

- There is exactly one node $n_{\text{root}} \in N$ such that n_{root} has no predecessors and there is a path from n_{root} to every other node $n' \in N$. The node n_{root} is called the *root node* of the tree.

- Every node $n' \in N$ other than the root node, $n' \neq n_{\text{root}}$, has exactly one direct predecessor.

A node without direct successors is called a *leaf node*, a node with at least one successor is called an *internal node*. The *depth* of a tree is the length of the longest path from its root node to one of its leaf nodes. □

Definition 12 (Binary Tree) A *binary tree* is a tree $\mathcal{T}$, where every internal node has exactly two direct successors. □

Definition 13 (Labeled Binary Tree) A *labeled binary tree* $\mathcal{T}$ is a binary tree, where each node n is represented by a tuple $(\mathcal{T}_{\text{left}}, l, \mathcal{T}_{\text{right}})$, such that $\mathcal{T}_{\text{left}}$ is the left subtree, l is the label of the node, and $\mathcal{T}_{\text{right}}$ is the right subtree. The label of a given node n is denoted as $l(n)$. □

3.1.4 Automata

Different kinds of automata are used across different working fields in computer science. In this thesis, the following definitions and terms will be used.

Definition 14 (Finite Automaton) A *finite automaton (FA)* $\mathcal{A}$ is a tuple $(S, \Sigma, \delta, s_0, F)$, where S is a finite set of states, Σ a finite input alphabet, $\delta : S \times \Sigma \rightarrow S$ the transition relation, $s_0 \in S$ the start state, and $F \subseteq S$ a set of accepting states.

An FA $\mathcal{A}$ is called *deterministic*, or *deterministic finite automaton (DFA)*, if for all $s \in S$ and for all $a \in \Sigma$, $\delta(s, a)$ has at most one element. □

For each FA, the transition relation δ defines a labeled graph $\mathcal{G} = (N, E)$, where the nodes in the graph are the states of the automaton, $N = S$, and the edges in the graph are derived from the transition relation, such that there is an edge $e = n_1 \xrightarrow{a} n_2 \in E$ if and only if there is a transition $\delta(s_1, a) = s_2$.

The *extended transition function* $\hat{\delta}(s, w) : S \times \Sigma^* \rightarrow S$ is defined as

$$\begin{aligned}(s, w \oplus a) &\mapsto \delta(\hat{\delta}(s, w), a) \text{ for } w \in \Sigma^+ \text{ and } a \in \Sigma \\ (s, a) &\mapsto \delta(s, a) \text{ for } a \in \Sigma \\ (s, \epsilon) &\mapsto s\,.\end{aligned}$$

Informally described, the extended transition function $\hat{\delta}(s, w) = s'$ computes the state s' of the automaton, where there exists a path from s to s' that is labeled by w.

Definition 15 (Accepted Language) A word w is *accepted* by an automaton $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$, if $\hat{\delta}(s_0, w) \in F$. Therefore, the *accepted language* $L(\mathcal{A})$ of the automaton consists of all words w that are accepted by the automaton, $L(\mathcal{A}) = \{w \mid \hat{\delta}(s_0, w) \in F\}$. □

Definition 16 (Finite State Machine) A *finite state machine (FSM)* is a quintuple $\mathcal{M} = (I, O, S, \delta, \lambda)$, where I is a finite input alphabet, O is a finite output alphabet, S is a finite set of states, $\delta : S \times I \rightarrow S$ is the transition function, and $\lambda : S \times I \rightarrow O$ is the output function. The notation $n_i \xrightarrow{a/u} n_j$ denotes an edge from node n_i to node n_j, which is labeled with input $a \in I$ and output $u \in O$. □

An FSM can be mapped to a DFA by combining the input and output alphabets of the FSM to serve as the input alphabet of the DFA, $\Sigma = (I \cup O)$. Every edge $n_i \xrightarrow{a/u} n_j$ labeled with an input $a \in I$ and an output $u \in o$ in the FSM is replaced by a sequence of two edges $n_i \xrightarrow{a} n_{\text{new}}$ and $n_{\text{new}} \xrightarrow{u} n_j$ in the DFA, the first labeled with the input and the second labeled with the output, $a, u \in \Sigma$. A similar concept is used by Holzmann [Hol91] in his definition of communicating finite state machines.

3.2 Automata in System Modeling

Automata of various kinds are widely used to model software and software systems. The concept of FSMs is especially used to model communication protocols, object-oriented software and state-based systems.

3.2.1 Communication Protocols

In communication systems, the message and signal interchange is mostly handled via interfaces. An interface defines a set of functions and procedures provided by a given software component. The protocol of an interface defines the use of those functions. Holzmann [Hol91] defines a protocol as a quintuple of the *service* provided by the protocol, some *assumptions* about the execution environment, a *vocabulary* or alphabet of messages, the *encoding* of each message, and the *procedure rules* of the message exchange. The procedure rules are described as an FSM over the alphabet of messages.

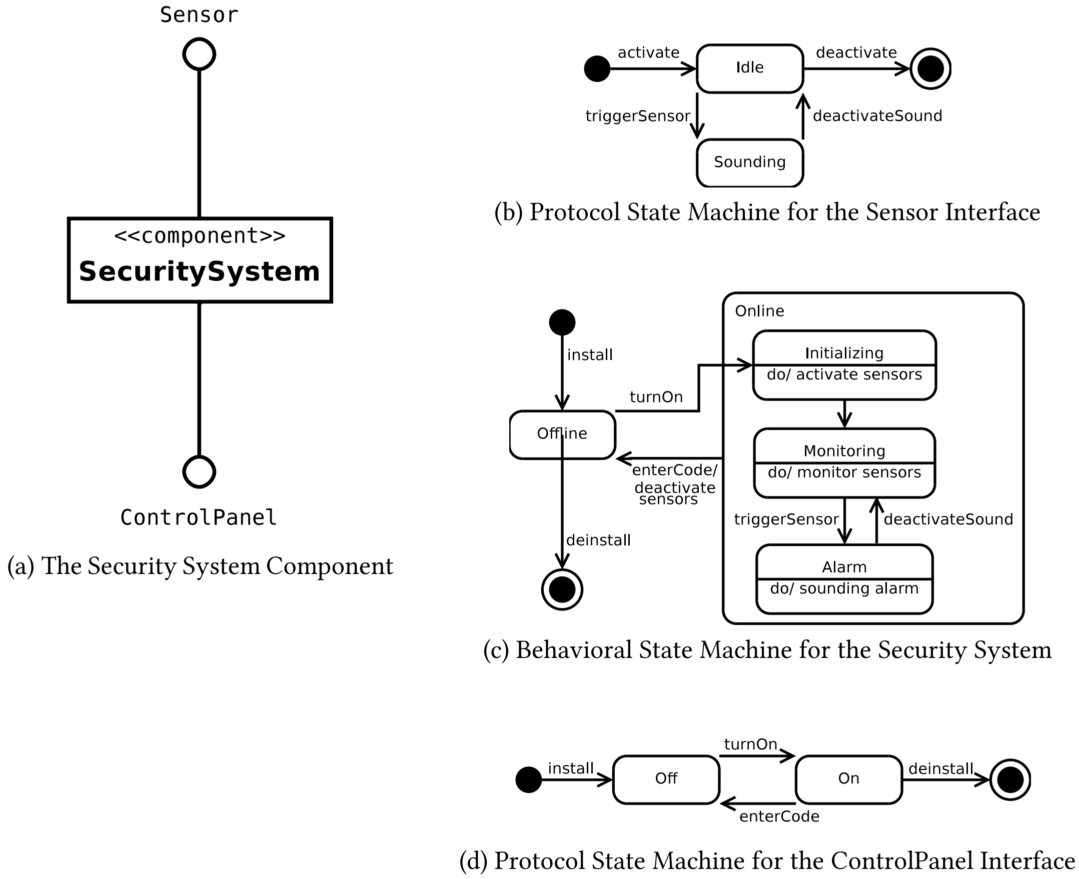
3.2.2 State-based Systems

According to Sommerville [Som06], a system is called *state-based*, if it responds to events differently over time, depending on its current state. Events may also cause transition from one state to another. A *reactive system* is driven by stimuli from its environment. A typical example for a state-based reactive system is that of a fire alarm that reacts to the detection of smoke and heat and accordingly changes state from “idle” to “alarm”.

3.2.3 State Machines in UML

In the *Unified Modeling Language (UML)*, a hierarchical type of state machines is used, which was fist introduced by Harel [Har87]. Although hierarchical state machines are more structured and thus more readable, they can be flattened and are therefore isomorphic to FSM as described before. UML distinguishes two types of state machines, behavioral state machines and protocol state machines. Behavioral state machines are used to model state-based components, e.g. classes, subsystems, or other components, whereas protocol state machines are used to describe the behavior of an interface to the component. Accordingly, a component may only have one associated behavioral state machine but arbitrarily many protocol state machines, as long as the protocol state machines are consistent with each other and the behavioral state machine. The protocol state machines may be visualized as a projection of the behavioral state machine onto a subset of the possible events, where the subset of events depends on the interface.

An example is given in Figure 3.1. On the left side, in Figure 3.1a, a security system component is displayed. The component provides two interfaces, a sensor interface and a control panel interface. On the right side, three state machines associated to the security system are depicted. Figure 3.1b shows a protocol state machine for the sensor interface. The protocol state machine describes the sequence of events on the sensor interface. Similarly, Figure 3.1d shows a protocol state machine for the control panel interface. Figure 3.1c shows the behavioral state machine for the security system. Where the protocol state machines focus on the events on the respective interfaces, the behavioral state machine shows the internal states of the security system and establishes a relation between the events on the different interfaces.



(a) The Security System Component

(b) Protocol State Machine for the Sensor Interface

(c) Behavioral State Machine for the Security System

(d) Protocol State Machine for the ControlPanel Interface

Figure 3.1: Protocol and Behavioral State Machines for the Security System

3.3 Traces

The term *trace* is used in different contexts with different constraints. For this thesis, the usages with respect to automata and to software systems are of interest. In the following, we will describe traces in the context of both areas and subsequently reconcile the description for further use in this thesis.

3.3.1 Automata View

In the automata view, a trace is similar to a word. The difference is that a word belonging to the accepted language of the automaton, $w \in L(\mathcal{A})$, necessarily starts in the start state s_o of the automaton and ends in an accepting state $s \in F$ of the automaton, whereas a trace may start and end at any state $s \in S$ of the automaton.

Definition 17 (Trace of an Automaton) A *trace* of an FA $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$ is a sequence of symbols $b_1, b_2, ...b_k$, $b_i \in \Sigma$, where there exists a path $s_1, s_2, ..., s_k, s_{k+1}$ in the automaton such that $\delta(s_i, b_i) = s_{i+1}$. □

A trace of an automaton is always ordered with respect to the sequence of the symbols, as the possible sequence of transitions of the automaton is defined by the transition relation.

3.3.2 System View

In the software system view, a trace is the observable behavior of a system. The term is also used to describe the execution logs of computer programs.

Definition 18 (Trace of a System) A *trace* is a sequence of events that a system has participated in. Therefore, a trace describes a possible behavior of the system at an arbitrary point in time. We distinguish

- *input events*, e.g. messages to the process, user inputs or signals,
- *output events*, e.g. outputs to a user interface or messages to another process and
- *internal events*, e.g. assignments. □

Definition 19 (Execution Trace) An *execution trace* is a trace that documents a complete run of a system, i.e. the execution trace starts in the start state of the system and ends in the final state of the system. □

For a system that may only behave sequentially, the ordering of the events in the trace implies an ordering of the activities of the system. For systems with parallel behavior, this is not the case, as events generated by different parallel subsystems may occur independently.

3.3.3 Reconciliation

In this thesis, we adhere to the automata view of traces. Therefore, we assume that each symbol in the trace depends on its predecessors. As a system can be modeled by an automaton, a trace of the system is also a trace of the system's model. For a system with parallel behavior, the model is flattened by computing the Cartesian product of the models of the parallel subsystems.

3.4 Testing

Sommerville [Som06] states that software testing has two main goals. On the one hand, successful tests are meant to increase confidence that the system correctly fulfills its requirements. On the other hand, testing aims at finding faults and defects in the software before it is shipped to the customers. In general, testing is an analytic means to assess and improve the quality of software. To reveal defects in the software, it is not necessarily required to execute the software. By examining the structure of the source code and computing characteristic figures, mostly referred to as *metrics* [FP98], anomalies can be detected without executing the source code. Quality assessment without executing the software is often referred to as *static testing*, in contrast to *dynamic testing* where the software is always executed.

The software to be tested is usually called a *system under test (SUT)*. The term might refer to software pieces of different size—a function, a program, a class, a component, or even a whole system.

Testing can be classified according to many different aspects, e.g. test methods, suitability for a level in the software engineering process, or suitability to a sort of software. In the following, we will take a closer look at test methods related to automata.

3.4.1 Test Cases

A test case is itself a software program. It sends stimuli to the SUT and receives responses from the SUT. Depending on the responses, the test case may branch out, and a test case can contain cycles to test iterative behavior. To each path through the test case's control flow graph, a verdict is assigned. In almost all test specification languages, the verdict *pass* marks an accepting test case and the verdict *fail* a rejecting test case. An *accepting* test case is a test case where the reaction of the SUT conforms to the expectations of the tester. This can also be the case, when an erroneous input is correctly handled by the SUT. A *rejecting* test case analogously is a test case where the reaction of the SUT violates its specification. Depending on the test specification language, there may be additional verdicts. The test specification language *Testing and Test Control Notation (TTCN-3)* [ETS07], for example, extends the usual verdicts *pass* and *fail* with the additional verdicts *none*, *inconc*, and *error*. The verdict *none* denotes that no verdict is set, *inconc* indicates that a definite assessment of the observed reactions is not possible, e.g. due to race conditions on parallel components, and *error* marks the occurrence of an error in the test environment.

As an SUT may be a modular system, it may also be distributed over a network, so that different parts of the SUT are located at different nodes of the network. The same applies to test cases. We distinguish *local* test cases, which consist of a single test component located on a single node, and *distributed* test cases, which consist of several test components that may be distributed over the network. To manage concurrent behavior among distributed test cases, the test components need to be coordinated.

For most SUTs, there is a collection of test cases, where each test case covers a certain behavioral aspect of the SUT. Such a collection of test cases for one SUT is called a *test suite*.

3.4.2 Structure-based Testing

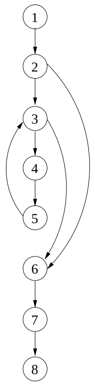
In *structure based testing*, also called *white box testing*, the test cases are derived from the source code of the SUT. To that end, a control flow graph is computed. The test cases then cover the control flow graph according to certain criteria. As structure based testing is derived from the structure of an underlying graph, the methods can be applied to any graph based structure [AO08, SLS07].

Definition 20 (Control Flow Graph) A *control flow graph* is a directed graph (N, E), where each node $n \in N$ represents a statement and each edge $e \in E$ represents a transition from one statement to another, including jumps. □

If a node in the control flow graph has more than one outgoing edge, the outgoing edges are also called *branches*. An example of a control flow graph for a small C program is given in Figure 3.2. The program code in Figure 3.2a is represented in the control flow graph in Figure 3.2b.

```
(1)  int hello(n) {
(2)    if (n < 10) {
(3)      for (i=0; i < n; i++) {
(4)        printf ("Hello World\n");
(5)      }
(6)    }
(7)    return 0;
(8)  }
```

(a) A "Hello World" Program



(b) Control Flow Graph

Figure 3.2: Program Code and Matching Control Flow Graph

Statement Coverage

A *statement coverage* of the control flow graph means that every node of the control flow graph is executed at least once in a test case. If there are silent edges, e.g. a condition without alternative, these might not be covered. For the example in Figure 3.2, a test case executing the path 1,2,3,4,5,3,6,7,8 satisfies a statement coverage, but the edge from node 2 to node 6 is never executed.

Branch Coverage

When every branch of the control flow graph is covered at least once in a test case, it is called *branch coverage*. In structural testing, this criterion is regarded as the minimal required coverage. For the example in Figure 3.2, in addition to a test case executing the path 1,2,3,4,5,3,6,7,8, another test case for the path 1,2,6,7,8 is needed. However, with this technique cycles in the graph are executed only once during one test run, so repetitive behavior is not checked.

Path Coverage

Path coverage of the control flow graph means that every possible path through the graph has to be executed. For the example in Figure 3.2, this would mean that in addition to the test cases for branch coverage, test cases repeatedly executing the cycle 3,4,5 are needed. There are two main problems to this criterion. First, successive branching in the control flow graph might be depending on each other, e.g. in the example, the **for** loop is only executed for values less than 10. In consequence, so some paths might not be possible to execute. Second, if there are cycles in the graph, the number of possible paths might be infinite, as the cycle can be traversed multiple times. To combine a repeated execution of cycles with a manageable number of test cases, there are a number of bounded path coverage criteria, e.g. the *boundary-interior coverage* which considers all paths where cycles are executed once, twice, or not at all.

The achieved coverage of a test suite is observed through *instrumentation* of the code. To this end, counters are inserted at strategic points, e.g. after conditional branches or cycles. The coverage percentage then is computed as the ratio of actually visited statements, edges, or paths to totally available statements, edges, or paths, respectively.

3.4.3 Specification-based Testing

In contrast to structure based testing, the test cases for *specification based testing*, also called *black box testing*, are derived from a specification of the SUT. Depending on the type of the specification, there are different kinds of black box tests.

Informal Specification

Often, there is only an informal or semi-formal specification of the SUT, e.g. a description of the functionality in natural language or a technical specification document. To measure the progress of the test, testing against an informal specification focuses on covering different aspects of the software.

One such aspect is the requirements of the SUT. In *requirements testing*, the test cases are selected to cover the different requirements documented in a technical specification document. For each requirement, a test case is designed to prove that the requirement has been met. In this method, a test case may cover many requirements, while some requirements may need several test cases.

Another aspect is the range of the input and output parameters of the SUT. In *equivalence partitioning*, input and output parameters are partitioned into equivalence classes. Test cases are selected so that for every valid and invalid equivalence class, at least one representative is tested. To minimize the number of test cases needed, test cases for different valid equivalence classes may be combined. However, every test case must only cover at most one invalid equivalence class, such that the observed reaction of the SUT can be definitely related to the invalid equivalence class. As errors often occur at the boundaries of equivalence classes, the equivalence class partitioning method can be complemented by *boundary value analysis*, where for every equivalence class, the values at the boundary and the neighboring values just inside and just outside the boundary are selected.

In *cause-effect graphing*, the description of the SUT is analyzed with respect to desired effects and the alleged causes. The cause-effect graph shows the relation between causes and effects using logical operators, i.e. negation, conjunction, and disjunction. From the graph, a decision table is derived. The test cases then are selected to cover the columns of the decision table.

In addition to the methods mentioned above, there are also a number of less formal testing techniques. To give some examples, a *smoke test* tries to find scenarios where the SUT crashes or seriously misbehaves; in *random testing*, the values for test cases are selected by random or according to a statistical distribution of the input values; and in *error guessing*, the tester draws on his experience to determine test cases that are likely to discover faults in the SUT.

State Based Test

If a formal specification is available, it is often in an automata-like formalism, such as Petri nets, process algebras, UML state machines, or another type of FSM. Test strategies for state based specifications usually include the generation of a state transition diagram, and test cases are then derived to cover the states and transitions of the diagram. In addition to state and transition coverage, the test can also focus on covering all events that cause a transition.

Object-Oriented Testing

Object-oriented testing often uses a similar strategy as state based testing. Binder [Bin99] suggests the *N+-Strategy* for testing modal classes, where every method of the class is tried in every state. The method should either show a correct output or a correct handling of the call, e.g. throw an exception. To achieve this, a so called *flattened regular expression (FREE) model* is constructed, which shows the state based behavior of the class depending on its methods and abstract states. Test cases are selected to cover the valid behavior of the class and also to check for invalid behavior. To cover the valid behavior, a state transition tree is constructed. Informally phrased, a state transition tree corresponds to an unrolled state transition diagram. For every path in the state transition tree, a test case is generated, which satisfies branch coverage of the FREE model. In addition to the state transition tree, a state response matrix is constructed, where possible and impossible transitions for each state are shown. For every impossible transition, another test case is generated to check whether the state does not change.

3.4.4 Protocol Conformance Testing

Protocol conformance testing is a special case of specification based testing suited to communication protocols (Section 3.2.1). For communication protocols, there often is a standardized specification, but the implementation of the protocols is up to the vendors of hardware and software. To ensure that hardware or software of different vendors interact correctly when using a protocol, each implementation of the protocol is tested against the standardized specification.

A protocol specification usually consists of two parts, a data part describing the type and structure of the data exchanged via the protocol, and a control part describing the behavior of the protocol, like the input/output behavior and constraints on the sequence of exchanged messages. The control part can often be described as an FSM. *Conformance* to a protocol is regarded as an equivalence relation between the behavior of the implementation and the specified behavior. An implementation conforms to a specification if for any input, the implementation produces the correct output [Hol91]. A number of methods have been proposed for protocol conformance testing. Most methods are based on the coverage of the protocol's behavior description, i.e. the protocol FSM, and are related to the structure based test generation methods used in white box testing. However, in white box testing, it is possible to observe the achieved coverage through code instrumentation. As the protocol implementation under test usually is only observable from the outside, in addition to the coverage of the FSM, some effort has to be taken to verify the target state of any tested transition.

3.4.5 Monitoring and Passive Testing

The notion of *passive testing* originates from the area of network management. While networks are growing larger and more complex, they are also growing more safety critical. The size and complexity of the networks inhibits extensive testing, while the safety critical properties increase the need for security.

Passive testing focuses on the observation of the inputs and outputs of the SUT during normal operation. The observed system traces are then compared to a specification of the network with the aim of detecting deviations [LNS+97, NSV03]. As passive testing does not interact directly with the SUT, it can also be applied when a direct stimulation of the SUT is too costly or otherwise inopportune. The term *monitoring* subsumes all methods needed for observing and logging the communication activities of the SUT.

3.5 Machine Learning

The aim of machine learning is to construct algorithms that are able to discover a system of rules from a collection of data. The algorithms thereby find a generalized representation of the available data, such that the generalized representation is able to classify data items correctly beyond the original input instead of remembering every single data item by heart.

Definition 21 (Input Space) The *input space* $\mathcal{X}$ is the set of objects the learner is interested in. An element $x \in \mathcal{X}$ of the target domain is called an *instance*. □

In this thesis, the input space contains all possible traces of an arbitrary SUT.

Definition 22 (Concept) A *concept* $c : \mathcal{X} \to \{0, 1\}$ defines a mapping from the input space $\mathcal{X}$ to $\{0, 1\}$, where $c(x) = 1$ indicates that x belongs to the concept and $c(x) = 0$ indicates that x does not belong to the concept. □

The *concept class* $\mathcal{C}$ is the set of all concepts that classify instances equivalently.

Definition 23 (Positive and Negative Example) An instance $x \in \mathcal{X}$ is called a *positive example* of the concept if $c(x) = 1$. If $c(x) = 0$, x is called a *negative example*. □

In this thesis, a concept c is the unknown DFA that models the SUT, and the concept class $\mathcal{C}$ is a set of all DFA that are equivalent to the model of the SUT. In order to learn the concept, a learning algorithm needs input in the form of a *training set*, or *sample*, of instances that describe the concept to learn.

Definition 24 (Training Set) A *training set* D to a concept c is a set of examples, where every training example is a pair $(x, c(x))$ of an instance x and the classification of this instance $c(x)$. The term $\mathrm{domain}(D)$ denotes all instances x where $(x, c(x)) \in D$. The set of positive examples, D_+, contains all instances x where $(x, 1) \in D$, analogously the set of negative examples, D_-, contains all instances x where $(x, 0) \in D$. □

In order to infer the unknown target concept from a training set, the learning algorithm tries to generalize the observations in the training set. To this end, the algorithm generates a *hypothesis*, which ideally should be identical to the concept.

Definition 25 (Hypothesis) A *hypothesis* $h : \mathcal{X} \rightarrow \{0, 1\}$ is a concept as per Definition 22. □

The *hypothesis class* $\mathcal{H}$ is the set of all possible hypotheses. As a hypothesis is a concept, the hypothesis class is also a class of concepts. In general, the hypothesis class $\mathcal{H}$ comprises the concept class $\mathcal{C}$, $\mathcal{H} \supseteq \mathcal{C}$. In this thesis, the hypothesis class is the set of all DFA.

Definition 26 (Consistency) A hypothesis $\mathcal{H}$ is called *consistent* with the training set D, if $h(x) = c(x)$ for all $x \in D$. □

The *version space* $\mathcal{VS}$ contains all hypotheses that are consistent with the training set.

The aim of learning is not only to reproduce the classification of the training examples correctly, but to use the learned concept to classify future examples. The *classification error* is the probability that a hypothesis disagrees with the concept on an instance, $h(x) \neq c(x)$. Typically, different candidate hypotheses h from the version space $\mathcal{VS}$ will make different classification errors on instances outside the training set. The problem of finding the hypothesis with the least classification error is known as *generalization*.

Learning a hypothesis from classified training examples is known as *supervised learning*. A learning algorithm that is restricted to a finite training set is called a *passive* or *offline* algorithm, whereas an algorithm that actively asks for the classification of instances is called an *active* or *online* algorithm. In this thesis, we only use techniques from the domain of supervised learning; therefore, we restrict this overview to the applied methods.

In the following, we introduce the two learning algorithms on which this thesis is based. Both algorithms generate a DFA from a classified example instances. The first algorithm (Section 3.6), a state-merging algorithm, needs a complete set of examples at the beginning, computes the maximal number of states from the examples, and then merges states according to a set of rules. The second algorithm (Section 3.7), a state-splitting automaton, assumes the minimal number of states and uses an external oracle to get the classification for selected instances. In both algorithms, the states of the DFA are identified by a trace leading from the start state of the DFA to the respective state. Therefore, words and states are interchangeable.

3.6 Automata Synthesis with State-Merging

One of the earlier approaches on the reconstruction of a DFA from a sample of accepting words, the set of positive examples D_+, and a sample of rejecting words, the set of negative examples D_-, was proposed by Biermann and Feldman [BF72]. Their algorithm constructs an automaton that accepts all words in D_+ and rejects all words in D_- based on the *Constraint Satisfaction Problem (CSP)* by guessing a state partitioning of the sample that preserves accepting and rejecting strings and setting the transition relation according to the sample. The crucial point of this algorithm is how to guess the state partitioning and to generate the transition function. Oncina and Garcia [OG92] have presented an efficient algorithm to solve that problem based on state-merging.

The main idea of the synthesis algorithm is to find a partitioning of the states in the positive sample D_+ that allows a more compact representation of the complete sample while preserving or even enlarging the expressiveness. The problem of finding such a state partitioning can be formulated as a guided search on all possible partitionings. However, as the possible search space is exponential in the number of states in the positive sample, an exhaustive search is not recommendable. Instead, the algorithm starts with a partitioning that is equal to all states in the prefix tree acceptor and then tries to merge pairs of states so that the rejecting words are still rejected by the reduced automaton.

The inference algorithm starts by constructing the prefix tree acceptor of the positive sample D_+.

Definition 27 (Prefix Tree Acceptor) Let D_+ be a positive sample from a regular language L. The *prefix tree acceptor* of D_+, $\mathrm{PT}(D_+)$ is defined as an FA

$$\mathrm{PT}(D_+) = (\mathrm{Pr}(D_+), \Sigma, \delta, \epsilon, D_+)\,,$$

where $\mathrm{Pr}(D_+)$ is the set of prefixes of D_+ (Definition 5) and δ is defined as $\delta(w, a) = wa$, $a \in \Sigma$ and $w, wa \in \mathrm{Pr}(D_+)$. □

The prefix tree acceptor can be seen as an automaton whose states are labeled by the prefixes of the positive sample, where the start state is marked with the empty word ϵ and the complete words $w \in D_+$ mark the accepting states.

In the each step, the algorithm tries to merge a pair of states of the prefix tree acceptor. Each time a merge is performed, the language accepted by $\mathrm{PT}(D_+)$ is increased, thereby possibly accepting a word form the negative sample D_-. In that case, the merge is reversed and the algorithm proceeds with the next pair of states. The algorithm finishes when there are no more pairs of states that can be merged.

It can be shown that if D_+ is a structurally complete sample, it is always possible to generate a deterministic automaton from the sample data. However, as the state-merging continually changes the data structure, the algorithm is not incremental, but needs the complete data at the beginning of the inference procedure.

For this thesis, the structural properties used in merging states are not applicable. Instead, in Chapter 5, we transfer the idea to the semantic context of test cases. After this introduction of automata synthesis, we now give the details of Angluin's learning algorithm.

3.7 Learning Finite Automata Using Queries

The technique of learning finite automata using a *minimally adequate teacher (MAT)* was first introduced by Dana Angluin [Ang87]. The algorithm has two parts, the *teacher*, which is an oracle that knows the concept to be learned, and the *learner*, who discovers the concept. The main idea of the learner is to successively discover the states of an unknown target automaton by asking the teacher whether a given word is acceptable to the target automaton.

3.7.1 Main Flow of the Learning Algorithm

We use Angluin's learning algorithm as described by Kearns and Vazirani [KV94]. The algorithm is structured into three parts:

- The *teacher* defines the knowledge about the target automaton $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$. The structure of the teacher itself is unknown to the algorithm. The teacher's knowledge is accessed via membership queries and equivalence queries.

- The *hypothesis automaton* $\mathcal{A}^H = (S^H, \Sigma, \delta^H, s_0, F^H)$ is the learner's guess at the target automaton at a given point in time. The hypothesis automaton has the same input alphabet Σ and the same start state s_0 as the target automaton.

- The *classification tree* is used to store the information about the states of the target automaton that has already been discovered.

Using the information from the teacher, the algorithm successively discovers the states of the target automaton. The main flow of the learning algorithm is quite simple. The algorithm starts with a *hypothesis automaton* that consists of a single state, which is labeled with the empty word. A membership query on the empty word determines whether the first state is accepting or not. Then, the hypothesis automaton is presented to the teacher, who either accepts it as equivalent to the target automaton or returns a counter-example. The counter-example is used to split one of the states of the hypothesis automaton, thereby generating a new state. Using the new state, a new hypothesis automaton is generated and put to test. These steps are repeated until the teacher is satisfied.

The information about the hitherto discovered states of the target automaton is stored in a data structure called *classification tree*, which is central to the learning algorithm. The classification tree is updated with the information from the counter-example and used to generate the new hypothesis automaton. A description in pseudo-code is presented in Algorithm 1; a graphical representation can be seen in Figure 3.3.

```
Result: DFA
Initialize the hypothesis automaton;
Initialize the classification tree;
while hypothesis automaton is not equivalent do
    get the next counter-example;
    use the counter-example to update the classification tree;
    update the hypothesis automaton;
end
```

Algorithm 1: The Learning Algorithm

3.7.2 Minimally Adequate Teacher

An MAT as defined by Angluin [Ang87] is able to answer two types of queries: *membership queries* and *equivalence queries*. A membership query determines whether a given word is part of the concept to be learned, i.e. the word is accepted by the target automaton. An equivalence query determines whether the constructed hypothesis automaton is equivalent to the target automaton.

Definition 28 (Membership Query) A *membership query* mq inquires whether a word $w \in \Sigma^*$ is accepted by the target automaton. The answer is either *yes* or *no*.

$$\mathrm{mq}(w) := \begin{cases} \text{yes} & \text{if } \hat{\delta}(s_0, w) \in F \\ \text{no} & \text{otherwise} \end{cases}$$

□

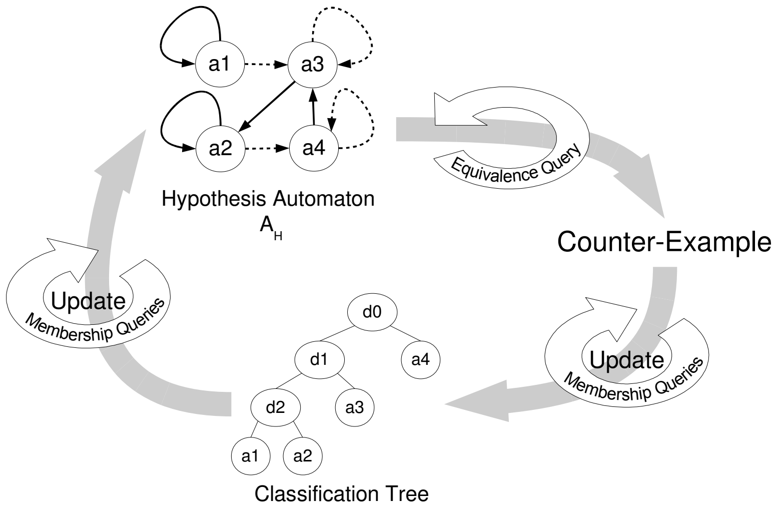


Figure 3.3: Repetitive Flow of the Learning Algorithm

Definition 29 (Equivalence Query) An *equivalence query* inquires whether the learned hypothesis automaton $\mathcal{A}^H$ is equivalent to the target automaton $\mathcal{A}$. The answer is a counter-example $c \in \Sigma^*$. If the empty word ϵ is returned as an answer, this means that no counter-example has been found. In this case, the automata are regarded as equivalent. □

3.7.3 The Learner

The second part of the learning algorithm, the learner, depends on the concept class to be learned. In our case, the concept class is a subset of the class of DFAs. A learner for this class needs a way of memorizing the already discovered states and a means to generate a new hypothesis automaton. According to Kearns [KV94], the discovered states are saved in a tree structure called *classification tree*, which is also used to generate the hypothesis automaton.

Every state in the target automaton $\mathcal{A}$ is identified by an *access string*. An access string is a word w_a that leads from the start state s_0 of the target automaton to the state $s \in S$ it identifies, such that $\hat{\delta}(s_0, w_a) = s$. For two distinct states s_1, s_2, $s_1 \neq s_2$, the access strings are also distinct. The access string of a state s is denoted by $w_a(s)$. The access string of the start state s_0 is the empty word ϵ.

Every pair of states (s_1, s_2) of the target automaton $\mathcal{A}$ is separated by a *distinguishing string*. A distinguishing string is a word w_d, where for each pair of access strings $w_a(s_1), w_a(s_2)$, with $s_1 \neq s_2$, exactly one of $w_a(s_1) \oplus w_d$ and $w_a(s_2) \oplus w_d$ is accepted by the target automaton. We denote the distinguishing string of two states s_1, s_2 by $w_d(s_1, s_2)$.

Definition 30 (Classification Tree) A *classification tree* $\mathcal{T}$ with respect to a target automaton $\mathcal{A}$ is a binary tree $(\text{left}, w, \text{right})$, where $w \in \Sigma^*$ is a word over the signal alphabet of the target automaton, left is the left subtree, and right is the right subtree. The word w at a given node n of the classification tree is denoted as $w(n)$, the left subtree is denoted as left(n) and the right subtree as right(n). A word that is stored at a leaf node of the classification tree is an access string w_a, and a word that is stored at an internal node of the classification tree is a distinguishing string w_d.

The classification tree has the following properties. The word w stored at the root node of the classification tree is the empty word ϵ. At every node n, in its left subtree left(n), words w' are stored where the concatenation $w' \oplus w(n)$ is not accepted by the target automaton, i.e. $\text{mq}(w' \oplus w(n)) =$ no. In the right subtree right(n), words are stored where the concatenation $w' \oplus w(n)$ is accepted by the automaton, i.e. $\text{mq}(w' \oplus w(n)) =$ yes. □

The access strings and distinguishing strings stored in the classification tree define a partitioning on the states of the target automaton, so the transitions of the hypothesis automaton $\mathcal{A}^H$ can be derived from the classification tree. The state s that is reached when an arbitrary word w is processed by the automaton, $s = \hat{\delta}(s_0, w)$, is computed by *sifting* the classification tree, which is shown in Algorithm 2. Starting at the root node of the classification tree, if the current node is an internal node, a membership query is made on the concatenation of w and the distinguishing string $w_d(n)$ stored at that node. If the word $w \oplus w_d$ is accepted, sifting continues in the right subtree of the current node, otherwise in the left subtree. When a leaf node is reached, the access string stored at this node defines the state s.

Generating a new hypothesis automaton

The transition with input $b, b \in \Sigma$, from an arbitrary state s in the hypothesis automaton, $s \in S^H$, is computed by sifting the word $w_a(s) \oplus b$. The word $w_a(s) \oplus b$ can be read as “starting in the start state, go to state s and then perform transition b” and thereby describes a path to the target state s_t, $\delta(s, b) = s_t$. This technique is used to compute the new hypothesis automaton in each cycle of the learning algorithm. The new hypothesis is initialized with the hitherto known states, i.e. all access strings stored at the leaves of the classification tree. Then, for every state s in the hypothesis automaton, the target

```
Data: An arbitrary string $w$
Result: An access string $w_a$
// Initialize
Start at the root of the classification tree;
while Current node is an internal node do
    Get the distinguishing string $w_d$ stored at the current node;
    Ask a membership query on $w \oplus w_d$;
    if $w \oplus w_d$ is accepted then
        Go to the right successor;
    else
        Go to the left successor;
    end
end
// The current node is a leaf
Get the access string $w_a$ stored ad the current node;
return $w_a$
```

Algorithm 2: Sifting the Classification Tree

states of the outgoing transitions b are determined by sifting $w_a(s) \oplus b$ for all $b \in \Sigma$. The pseudo-code for the generation of a new hypothesis automaton is given in Algorithm 3.

```
Initialize the states of the hypothesis automaton with the leaves of the classification
  tree;
for every state $s$ in the hypothesis automaton do
    for every signal $b$ in the signal alphabet do
        $s_t = \text{sift}(w_a(s) \oplus b)$;
        Set a transition $e$ from $s$ to $s_t$;
        Label the transition $e$ with $b$;
    end
end
```

Algorithm 3: Generating a New Hypothesis Automaton

Updating the Classification Tree

When a hypothesis automaton is queried for equivalence, as long as the learning process is not completed the answer is a counter-example. This counter-example is used to split one of the hitherto known states into two sub-states. The algorithm to update the classification tree works in two phases (Algorithm 4).

In the first phase, we look for a state that can be split. For this, we check the prefixes p_i of length i of the counter-example, starting with the shortest $i = 1$, by sifting (Algorithm 2) the prefix, $\mathrm{sift}(p_i)$, and by processing the prefix on the hypothesis automaton $\hat{\delta}(s_0, p_i)$. Let p_{diff} be the shortest prefix where $\mathrm{sift}(p_{\text{diff}})$ and $\hat{\delta}(s_0, p_{\text{diff}})$ differ. Then, the prefix $p_{\text{match}} = p_{\text{diff}}$ marks the longest prefix where the results of sifting and hypothesis match and determines the state to be split as $s_{\text{split}} = \mathrm{sift}(p_{\text{match}})$.

In the second phase, s_{split} is replaced by an internal node s_{int} with two successors. The label $w_d(s_{\text{int}})$ of the internal node, a distinguishing string, is determined from the counterexample and the distinguishing strings as

$$w_d(s_{\text{int}}) = p_{\text{diff}}[\text{diff}] \oplus w_d(\mathrm{sift}(p_{\text{diff}}), \hat{\delta}(q_0, p_{\text{diff}}))\ ,$$

where $p_{\text{diff}}[\text{diff}]$ denotes the last signal in the prefix p_{diff} (Definition 1). One of the successors is marked with the access string of s_{split}, $w_a(s_{\text{split}})$. The other successor is marked with the prefix p_{match}. The successor s, where $w_a(s) \oplus w_d(s_{\text{int}})$ is accepted by the automaton is inserted to the right.

It is noteworthy that the thus obtained deterministic automaton will always be minimal, even if the training data, i.e. the teacher that determines the counter-examples, is not.

3.8 Summary

In this chapter, we have introduced the necessary background for our research. As particularly well-known theoretical concepts are often used with slight differences, we have provided a collection of definitions and concepts as a theoretical foundation for our own work. In order to introduce the terminology of our application area, we have given an overview on modeling techniques, the concept of traces and the principles of software testing. Lastly, we have described the two learning approaches that form the core of our learning procedure. The state-merging approach uses a preferably complete collection of positive examples of a language recognized by an automaton to reconstruct the automaton by merging states with identical suffixes. In contrast, Angluin's learning algorithm discovers the states of the hidden target automaton by querying an oracle for the classification of example strings. In the subsequent chapters, we will tailor Angluin's learning algorithm to the domain of testing, with the help of optimizations derived from the state-merging approach.

```
Data: counter-example
// Initialization
i = 0;
// Find the state to split
repeat
    $i = i + 1$;
    Compute the current prefix $p_i$ as the first $i$ letters of the counter-example;
    Sift the current prefix: $\mathrm{sift}(p_i)$;
    Execute the current prefix on the hypothesis automaton: $\hat{\delta}(s_0, p_i)$;
until $\mathrm{sift}(p_i) \neq \hat{\delta}(s_0, p_i)$;
// Replace the state to split by an internal node with two leafs
The state to split is marked with the access string $\mathrm{sift}(p_{i-1})$;
The access string of the new state is $p_{i-1}$;
Get the distinguishing string $d$ of the least common predecessor from the differing
  states $\mathrm{sift}(p_i)$ and $\hat{\delta}(s_0, p_i)$;
Compute a new distinguishing string $w_{d_{\text{new}}}$ as concatenation of the last letter of $p_i$
  and $d$, $p_i[i] \oplus d$;
Ask a membership query on the string $p_{i-1} \oplus w_{d_{\text{new}}}$, the concatenation of the access
  string of the new state and the new distinguishing string;
if $p_{i-1} \oplus w_{d_{new}}$ is accepted then
    Move the old state $\mathrm{sift}(p_{i-1})$ to the left successor;
    Insert the new state $p_{i-1}$ as the right successor;
else
    Insert the new state $p_{i-1}$ as the left successor;
    Move the old state $\mathrm{sift}(p_{i-1})$ to the right successor;
end
Mark the parent of $p_{i-1}$ and $\mathrm{sift}(p_{i-1})$ with the new distinguishing string $w_{d_{\text{new}}}$;
```

Algorithm 4: Updating the Classification Tree

4 Adaptation of Angluin’s Algorithm to Learning from Test Cases

Our aim is to synthesize a *deterministic finite automaton (DFA)* from test cases. Learning algorithms provide a means of automating this task, while at the same time allowing for some amount of external regulation. Angluin’s learning algorithm especially has the advantage of generating a minimal DFA, while the requirements on the input data are already similar to the structure of a test suite.

The main idea of Angluin’s learning algorithm is to successively discover the states and transitions of the target automaton by querying an oracle, the teacher. Whether the teacher is a database, an automaton, or a human being makes no difference, as long as the requirements of a *minimally adequate teacher (MAT)* are met. In the context of Angluin’s learning algorithm, an MAT has to be able to answer two types of queries:

- *membership queries* discover whether a given sequence is accepted by the target automaton,

- *equivalence queries* ask whether the hypothesis automaton already matches the target automaton. If the hypothesis automaton differs from the target automaton, the teacher answers by giving a counter-example.

A detailed description of Angluin’s learning algorithm can be found in Section 3.7.

As the learning algorithm generates a DFA, the output of the algorithm is already well-suited to our needs. Therefore, we use the learner part of the algorithm as it is. The main adaptation for learning a finite automaton from test cases is to define a suitable MAT. We suggest a twofold approach [WGTZ08]. On the one hand, a representation of test cases has to be found that can be used as input data to the learning algorithm. On the other hand, the query mechanisms of the algorithm, i.e. membership queries and equivalence queries, have to be adapted to the properties of test cases. Clearly, there are interdependencies between the two adaptations.

In the following, we will present our adaptation of Angluin's algorithm. First and foremost, this means an analysis of Angluin's learning algorithm to show where adaptation is necessary. To this end, the learning algorithm is analyzed with respect to the generated automaton and to the requirements regarding the input traces. Subsequently, we will introduce two adaptations of the MAT and provide an example and analysis for each. The first adaptation describes the fundamental redefinition of the queries in the context of test cases. Based on the analysis of the first adaptation, the second adaptation defines an optimization to exploit explicit information on cyclic behavior for the learning process.

4.1 Analysis of the Learning Algorithm

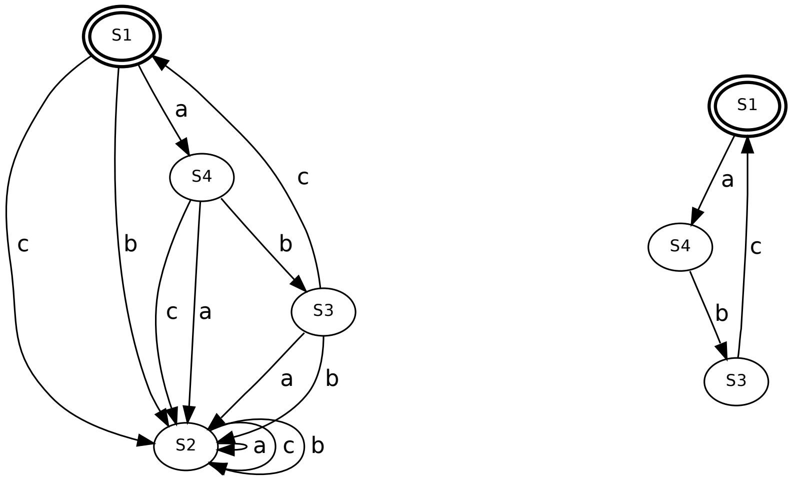
To judge the suitability of an algorithm to a certain application, three parts of the algorithm have to be taken into account. First of all, there is the output of the algorithm. Is the output suitable to our need? Second, we have to regard the input of the algorithm. Can we meet the requirements of the algorithm? Only when those two questions are solved we can decide whether the adaptation of the algorithm is feasible and which adaptations are necessary.

4.1.1 Output of the Algorithm: A Deterministic Finite Automaton

The automaton generated by the learning algorithm is a DFA, i.e. there is only an input alphabet, and no output alphabet further than $\{0, 1\}$, which indicates the differentiation between accepting and non-accepting states. Usually in system modeling more advanced automata are used, e.g. *finite state machines (FSMs)* or even *extended finite state machines (EFSMs).* However, Holzmann [Hol91] uses a linearized variation of FSMs for his communicating finite state machines, on the grounds that simple transitions with only one input are better adapted to synchronization.

Another property of system modeling is that automata describing system models are in most cases not fully specified, i.e. only the signals of interest are shown in the model. One possible interpretation is to consider the missing transitions in an under-specified automaton as self-loops, meaning that any inopportune signal is ignored. In contrast, the learning algorithm always generates a fully specified automaton. In this automaton, all rejected transitions are routed into the same non-accepting state, which is further referred to as the *global fail state.* The global fail state is used as a drain for all rejecting examples. From the properties of the learning algorithm (Section 3.7, we can deduce that the global fail state is always the state that is stored at the leftmost leaf of the classification tree. The learned automaton can therefore be seen as a “strict” version of the target automaton, which rejects all inopportune signals. As even in small automata many transitions drain into the global fail state, the learned automata are complicated and hard to read. However, as the global fail state can be clearly identified, it is possible to remove it and the draining transitions for better readability.

Figure 4.1 shows two versions of a simple automaton, which accepts all strings (a,b,c)*. The automaton starts and ends in the state S1. In Figure 4.1a, the automaton is shown as a strict version, where the rejected transitions are explicitly routed into the state S2. Figure 4.1b shows the same automaton, where the global fail state has been removed.



(a) Strict Automaton With Global Fail State

(b) Automaton Without Global Fail State

Figure 4.1: Elimination of the Global Fail State

The output of the learning algorithm is always a deterministic, minimal, and flat DFA, where every possible interleaving sequence of parallel behavior and all data values are represented explicitly. In particular, the simple DFA will always need more states than more expressive models. While not being a problem for small examples, this property of the learning algorithm will result in the well known state space explosion problem with increasing number of inputs and data values. However, on the whole it can be shown that simple DFAs are isomorphic to more advanced automata models, so for a first cut at the learning of a model from test cases, the learning algorithm can be used as it is.

4.1.2 Input of the Algorithm: Positive and Negative Examples of the Target Automaton

Angluin's algorithm receives its input via the query mechanism of the MAT, where the membership query assesses the acceptance of a word and the equivalence query assesses the correctness of the inferred automaton and generates counter-examples. Both queries work on the sample set, which consists of positive and negative examples of the language to be learned, i.e. words that are accepted by the automaton and words that are rejected by the automaton. There are two types of aspects to be considered. On the one hand, the properties of the words themselves are important when mapping test cases to inputs of the learning automaton. On the other hand, also the structure of the complete sample has to be taken into account, i.e. all words that are queried by the automaton. When learning from test cases, the structure of the sample relates to the test suite and the coverage of the *system under test (SUT)* rather than to individual test cases.

Let us first consider the properties of the example words. First of all, the learning algorithm relies on a single starting state of the target automaton. The reason for this is the state discovering process, where states are named by their access string, which describes the path from the starting state to the thusly named state. As a consequence, the learning algorithm simply cannot distinguish different starting states, and will accordingly map all starting states into one. Test cases, on the other hand, are not required to start in the starting state of the SUT, as long as they start in a defined state that can be reliably established. It is, however, quite usual to prefix a test case by a *preamble*, which drives the test case from such a defined state of the SUT into the state of interest to the test case. By electing one of the defined states as the overall starting state and prefixing all test cases with according preambles, we can make sure that all test cases start in the same unique starting state.

As the learning algorithm generates a DFA, there is only one input alphabet and the output alphabet is restricted to *accept* and *reject*. Most test specification languages, and therefore also the test cases, differentiate at least between signals sent to the SUT, signals received from the SUT, and internal computations of the test case itself. The setting of a verdict, which determines whether a certain behavior should be accepted by the SUT, is usually part of these internal computations. While a mapping between the alphabets of the test cases and those of the learning algorithm is certainly necessary, we have to be careful not to mix up "input" and "output" of test cases and the learning algorithm, as those terms denote different things for each concept. In addition, we have to consider ports, as there are software systems that show the same behavior on different ports, e.g.

when the same interface is implemented for two hardware architectures. To map the vocabulary of the test case to the learning algorithm, we assume the following.

- Inputs and outputs of the test cases, i.e. signals received from the SUT and sent to the SUT, are both mapped to the input alphabet of the hypothesis automaton. If there is a signal a that is both sent to and received from the SUT, we will regard it as two items in the algorithm's input alphabet, namely a_{in} and a_{out}, or $!a$ and $?a$ for short.
- Ports are regarded as a part of the signal definition, so a signal a that can be sent or received on more than one port will make an item in the algorithm's input alphabet for every port it is sent or received on. We will write port!signal or port?signal.
- The setting of verdicts in the test case is mapped to the output alphabet of the hypothesis automaton, so that behavior of the SUT which is rated by a *pass* verdict is regarded as a word that should be accepted by the hypothesis automaton and behavior of the SUT which is rated by a *fail* verdict is regarded as a word that should be rejected by the hypothesis automaton.
- As we are interested in learning the SUT's external behavior, we ignore all other internal actions of the test cases.

Lastly, the learning algorithm uses words, which are by definition linear sequences of items from an alphabet. In contrast, test cases in general are software programs themselves and may have branching or cyclic behavior. Therefore, every test case maps to a number of words for the learning procedure, one word for every possible path through the test case's behavior.

Given these consideration, we define the notion of *test case execution trace* as input to our learning procedure.

Definition 31 (Test Case Execution Trace) A *test case execution trace* t is a tuple $(w, v(w))$ where w is an execution trace of a test case and $v(w) \in \{\textit{pass}, \textit{fail}\}$ is the verdict that this execution trace is assigned by the test case. In the following, we will denote the execution trace of a given test case by $w(t)$ and the expected verdict as $v(w(t))$. □

As explained in Section 3.3, a trace can be interpreted as a word. Therefore, an accepting execution trace t of a test case T, where $v(w(t)) = \textit{pass}$, maps to a word that should be accepted by the hypothesis automaton. Accordingly, a rejecting execution trace of a test case, where $v(w(t)) = \textit{fail}$, maps to a word that should not be accepted by the hypothesis automaton. As every test case may contain a number of possible execution traces, the collection of all test case execution traces belonging to a given test case T will be denoted by $\mathrm{traces}(T)$. Analogously, we will denote the complete collection of all test case execution traces of all test cases belonging to a given SUT, i.e. the traces of the complete test suite $\mathcal{TS}$, as $\mathrm{traces}(\mathcal{TS})$. In either case, the trace collection $\mathrm{traces}(x)$ contains both accepting and rejecting test case traces.

Besides the mapping of the single test cases, we also have to take into account the structure of the whole test suite. Angluin's learning algorithm relies on examples that are classified as "accepting" and "rejecting". As described above, we have mapped the verdicts *pass* and *fail* onto accepting and rejecting examples, respectively. However, test specification languages usually have at least one additional verdict, *none*, to indicate that no verdict has yet been set on a given path, and sometimes also verdicts indicating errors in the test environment like the *Testing and Test Control Notation (TTCN-3) error*. While the mapping of the verdicts *pass* and *fail* is obvious, the handling of additional verdicts has to be considered.

Let us take a closer look at the two query mechanisms. The aim of the equivalence query is to establish that the learned automaton actually represents all the examples. Essentially, this is also the aim of a test suite, which wants to establish conformance between the SUT and the specification. For this assessment, the test case traces with verdicts *pass* and *fail* suffice. Therefore, for the equivalence query we can simply ignore additional verdicts.

In contrast, the membership is used to establish the acceptability of single traces. The assessment of the queried trace is then used to route a transition in the hypothesis graph or to determine where in the classification tree a new state hast to be added. While a wrong transition in the hypothesis automaton can easily be remedied in the next iteration of the learning algorithm, a new state added in the wrong place in the classification tree affects all further iterations, leading to wrong transitions in the hypothesis automata and finally to an erroneous result. For this reason, it is important to answer membership queries correctly, even if the trace is not explicitly assessed by a *pass* or *fail* verdict. There are two possible approaches, known as *closed world approach* and *open world approach*. The most reliable, but also most restrictive, approach is the closed world approach, which is to accept only words which can be mapped on a test case with the verdict *pass* and to reject anything else. The contrary concept, the *open world* approach, assumes that not everything is known about the system to learn. In consequence, only words that are explicitly marked as *fail* by a test case can be rejected, whereas words for which no verdict can be determined cannot be assessed. As this leads to uncertainties in the learning procedure, we will adhere to the closed world approach.

The closed world approach's strategy to reject everything which is not explicitly allowed leads to another question regarding the structure of the test suite. If we only accept what is definitely marked as acceptable, then we need to have all acceptable traces in our test suite. In the worst case, this means that path coverage of the possible behavior of the SUT is needed.

4.2 Learning from Finite Traces

Based on the mapping of test cases onto the test case execution traces, we define an adaptation of the MAT in Angluin's algorithm. Subsequently, we analyze the complexity of our approach with respect to the size of both the generated automaton and the test suite. We demonstrate the functionality of our adaptation by way of a small and therefore comprehensible example.

4.2.1 Adaptation of the Teacher

The most important mechanism of the learning algorithm is the membership query, which determines the acceptability of a given behavior. In our case, the behavior of the software and thus of the target automaton is defined by the test cases. Since in the closed world scenario, the test cases are our only source of knowledge, we assume that the test cases cover the complete behavior of the system. In consequence, we state that every behavior that is not explicitly allowed must be erroneous and therefore has to be rejected, i.e. *rejected* $\equiv \neg$*accepted*. Thus, the membership query is redefined as follows:

Definition 32 (Membership Query on Test Cases) A word w is *accepted* by the target automaton if it matches an accepting test case execution trace t in the trace collection of the test suite, otherwise it is rejected.

$$\mathrm{mq}(w(t)) := \begin{cases} \text{accept} & v(w(t)) = \textit{pass} \\ \text{reject} & \text{otherwise} \end{cases}$$

□

Likewise, the equivalence query can be redefined as an execution of the test suite against the hypothesis automaton.

Definition 33 (Equivalence Query on Test Cases) The hypothesis automaton $\mathcal{A}^H = (S^H, \Sigma, \delta^H, s_0, F^H)$ is *equivalent* to the target automaton, if for every test case execution trace t in the traces of the test suite, $t \in \text{traces}(\mathcal{TS})$, the processing of its execution trace $w(t)$ on the automaton is accepted or rejected as specified by the test verdict $v(w(t))$.

$$\text{eq}(\mathcal{A}^H) := \begin{cases} \text{yes} & \begin{aligned} \forall t \in \text{traces}(\mathcal{TS}) : (\hat{\delta}^H(s_0, w(t)) \in F^H) \Leftrightarrow (v(w(t)) = \text{accept}) \wedge \\ (\hat{\delta}^H(s_0, w(t)) \notin F^H) \Leftrightarrow (v(w(t)) = \text{reject}) \end{aligned} \\ \text{no} & \text{otherwise} \end{cases}$$

The first test case execution trace that does not reproduce its verdict is returned as a counter-example. □

4.2.2 Example

To illustrate the learning from finite traces, let us pose a simple example as shown in Figure 4.2. The system describes a small coffee machine. After inserting money and requesting the coffee, the machine outputs a cup of coffee. The coffee machine has only one port, cm, where inputs and outputs are recorded. Each accepted interaction with the coffee machine starts and ends in state S0.

For the coffee machine example, we generate the test case execution traces shown in Table 4.1, following a boundary-interior coverage of the target automaton. For the beginning of the learning algorithm, we also need a classification of the empty trace, which is stated as test case 1 in Table 4.1. The classification of the empty trace is needed to correctly determine whether the starting state of the automaton is an accepting or rejecting state.

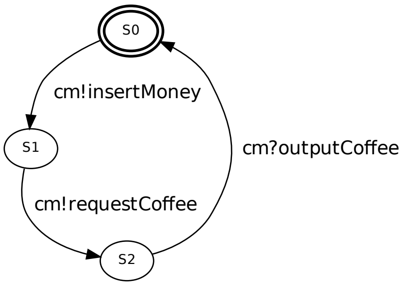


Figure 4.2: A Small Coffee Machine

ID	Test Case Execution Trace	Verdict
1	ϵ	*pass*
2	cm!insertMoney, cm!requestCoffee, cm?outputCoffee	*pass*
3	cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm!insertMoney, cm!requestCoffee, cm?outputCoffee	*pass*
4	cm!insertMoney, cm!insertMoney	*fail*
5	cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm!insertMoney, cm!insertMoney	*fail*
6	cm!insertMoney, cm?outputCoffee	*fail*
7	cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm!insertMoney, cm?outputCoffee	*fail*
8	cm!insertMoney, cm!requestCoffee, cm!insertMoney	*fail*
9	cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm!insertMoney, cm!requestCoffee, cm!insertMoney	*fail*
10	cm!insertMoney, cm!requestCoffee, cm!requestCoffee	*fail*
11	cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm!insertMoney, cm!requestCoffee, cm!requestCoffee	*fail*
12	cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm!requestCoffee	*fail*
13	cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm!requestCoffee	*fail*
14	cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm?outputCoffee	*fail*
15	cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm?outputCoffee	*fail*

Table 4.1: Test Case Execution Traces for the Coffee Machine

From the generated test case execution traces, we reconstruct the original automaton. First, the hypothesis automaton is initialized. For the coffee machine, as the empty trace is accepted, the initial state is an accepting state. The initial hypothesis automaton is shown in Figure 4.3. In all graphs, the node representing the starting state is shown with a bold frame, while nodes representing accepting states are shown with a double frame. If starting and accepting state are represented by the same node, then this node is shown with a bold double frame.

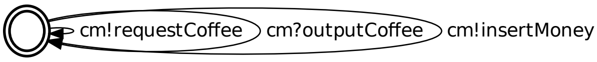
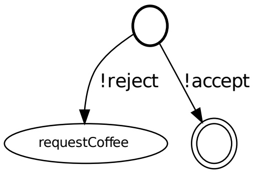
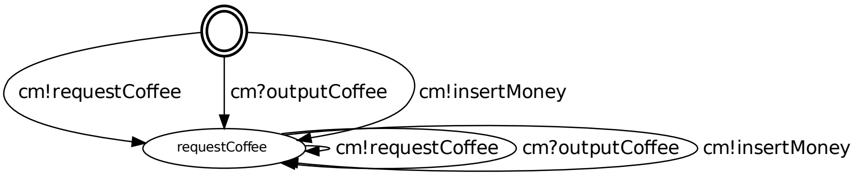


Figure 4.3: Initial Hypothesis Automaton

A first counter-example is generated, the trace cm!requestCoffee with the verdict *fail*, and the classification tree is initialized (Figure 4.4a). As the start state of the automaton is an accepting state, the first counter-example is the shortest rejecting trace in the test suite. From the classification tree, a new hypothesis automaton is generated (Figure 4.4b). The new hypothesis contains the newly discovered state requestCoffee, where the new state is labeled according to its access string in the classification tree. The new state requestCoffee also serves as the global fail state (Section 4.1.1). This completes the first iteration of the learning algorithm (Figure 4.4).



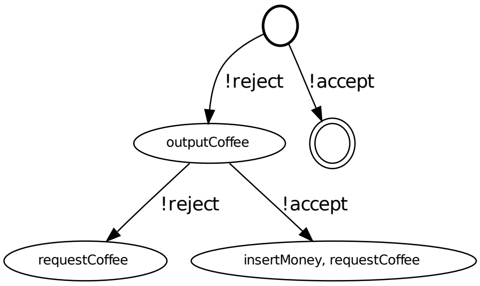
(a) Initial Classification Tree



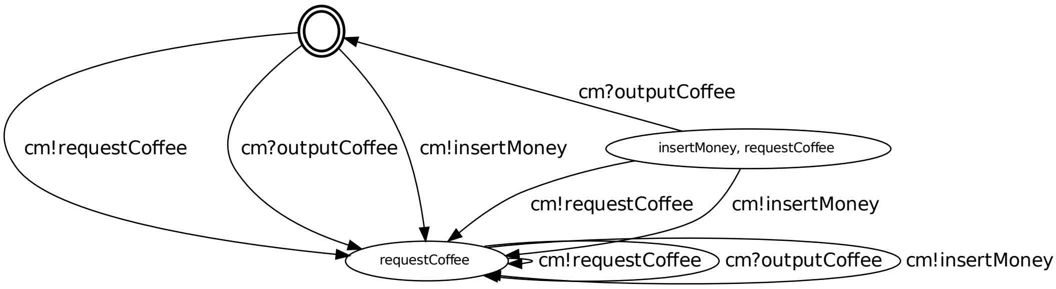
(b) New Hypothesis Automaton

Figure 4.4: First Iteration of the Learning Procedure

As the new hypothesis automaton (Figure 4.4b) is not equivalent to the target automaton, a new counter-example c is generated, cm!insertMoney, cm!requestCoffee, cm?outputCoffee with verdict *pass*, and the classification tree is updated (Figure 4.5a). Again, a new hypothesis automaton is generated (Figure 4.5b), and the second iteration is complete (Figure 4.5). As only a prefix of the counter-example is used to find the new state insertMoney,requestCoffee, the prefix of length 2 $p_2(c)$ = cm!insertMoney, cm!requestCoffee, the new hypothesis automaton still classifies the complete counter-example incorrectly.



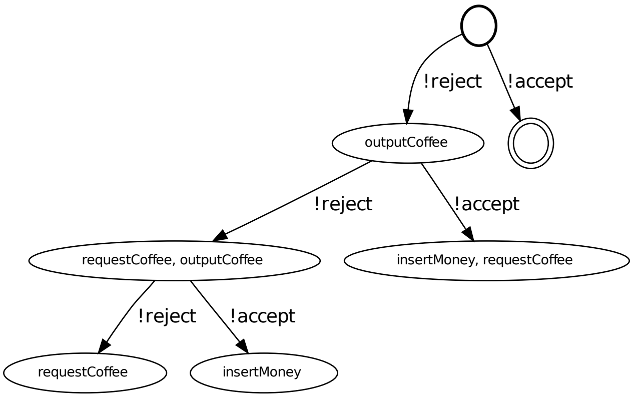
(a) Classification Tree



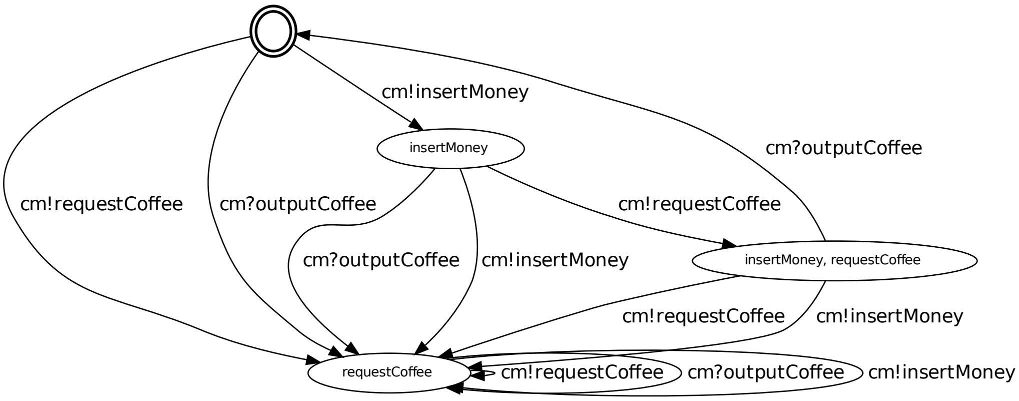
(b) Hypothesis Automaton

Figure 4.5: Second Iteration of the Learning Procedure

As the hypothesis automaton is still not equivalent, we get another counter-example, cm!insertMoney, cm!requestCoffee, cm?outputCoffee, *pass*. The new counter-example is the same as the last one, as we have only used a prefix of the counter-example in the second iteration and the complete counter-example is still not classified correctly. Following the same procedure as before, we update the classification tree (Figure 4.6a) and build a new hypothesis automaton (Figure 4.6b).



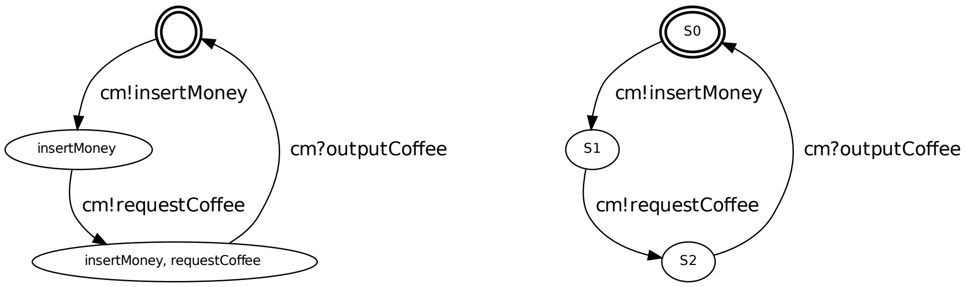
(a) Classification Tree



(b) Hypothesis Automaton

Figure 4.6: Third Iteration of the Learning Procedure

This completes the learning procedure, as the last hypothesis automaton correctly reproduces all traces in the test suite as listed in Table 4.1. Figure 4.7a shows the final automaton without the global fail state, which matches the original automaton. For easier comparison to the automaton used to generate the test traces, the original coffee machine automaton (Figure 4.2) is repeated in Figure 4.7b. With the sole exception of the state labeling, both automata are identical.



(a) Learned Automaton Without Global Fail State

(b) Target Automaton

Figure 4.7: Elimination of the Global Fail State

4.2.3 Query Complexity in Relation to the Size of the Test Suite

As our adaptation of Angluin's learning algorithm concerns the query mechanisms, this is the point where the complexity is influenced. At the same time, the requirements of the respective query also influence the structure and the complexity of the test suite.

Query Complexity

For our current definition of the MAT, the complexity of both query mechanisms depends on the size of the test suite, i.e. the number and length of the test case execution traces. The detailed analysis of our query mechanisms is as follows.

Theorem 1 (Estimation of the Number of Membership Queries) *The number of membership queries is $O(|\Sigma| \cdot |S_{target}|^3 + l \cdot |S_{target}|^2)$, where $|\Sigma|$ is the size of the signal alphabet, l is the length of the counter-example, and $|S_{target}|$ is the number of states in the target automaton.*

Proof Membership queries are used in the generation of a new hypothesis automaton and in the updating of the classification tree. When generating a new hypothesis automaton, for every state of the automaton and every signal in the signal alphabet, the target state of the corresponding transition has to be determined. This is achieved by appending the signal b to the access string w_a of the current state and sifting the resulting string $w_a b$ through the hypothesis automaton. As soon as we reach a leaf, we have found the target state. As every sifting requires in the worst case as many membership queries as the current depth of the classification tree, we get $|S_{\text{hypothesis}}| \cdot |\Sigma| \cdot d$ membership queries, where $|S_{\text{hypothesis}}|$ is the number of states in the current hypothesis automaton, $|\Sigma|$ is the size of the signal alphabet, and d is the current depth of the classification tree. We know that the leaves of the classification tree describe the states of the current hypothesis automaton. Therefore, as the classification tree is not necessarily balanced, the current depth of the classification tree is at most one less than the number of states in the current hypothesis automaton, and in consequence we can simplify our upper bound as $|S_{\text{hypothesis}}|^2 \cdot |\Sigma|$.

When a counter-example is processed, we sift every prefix of the counter-example, beginning with the shortest, until a difference to the current hypothesis automaton is found. In the worst case, the difference is only found when sifting the whole counter-example. As before, for every sifting, at each level of the classification tree a membership query is generated until a leaf is reached. Therefore, the number of membership queries in the processing of the counter-example is at worst $l \cdot d$, where l is the length of the current counter-example and d is the depth of the current classification tree. Again, as the maximum depth of the classification tree depends on the number of states of the hypothesis automaton, we can give the estimation also as $l \cdot |S_{\text{hypothesis}}|$.

In every iteration of the learning algorithm, a new hypothesis automaton is generated and a new counter-example is processed. As every hypothesis automaton has at most the same number of states as the target automaton, we use the number of states in the resulting automaton, $|S_{\text{target}}|$, as an upper bound for the number of states in the hypothesis automata. Therefore, in every iteration, the learning algorithm generates $O(|\Sigma| \cdot |S_{\text{target}}|^2 + l \cdot |S_{\text{target}}|)$ membership queries.

As the learning algorithm finds a new state in every iteration, the number of iterations linearly depends on the states in the resulting automaton. Therefore, we can state that during the complete learning algorithm, the number of membership queries is $O(|S_{\text{target}}| \cdot (|\Sigma| \cdot |S_{\text{target}}|^2 + l \cdot |S_{\text{target}}|))$, which can be expanded to $O(|\Sigma| \cdot |S_{\text{target}}|^3 + l \cdot |S_{\text{target}}|^2)$. ■

The proof of Theorem 1 shows that the number of membership queries asked during the learning procedure depends on the structure of the target automaton, i.e. the size of the target automaton and the size of the signal alphabet. The only influence of the test suite on the number of membership queries is the length of the counter-example, as this is related to the length of the traces in the test suite. However, as the learning algorithm always uses the smallest differing prefix of the counter-example (Section 8), this influence is of minor importance.

In contrast, the runtime complexity of the membership query clearly depends on the size and structure of the test suite. As membership queries are answered by finding a corresponding test case execution trace, each membership query means a search on the test suite. Therefore, the runtime complexity of the membership query depends on the complexity of a search on the test suite execution traces, $\mathrm{traces}(\mathcal{TS})$. In consequence, for larger test suites, an efficient search strategy has to be implemented.

While the size of the test suite only marginally influences the complexity of the membership query, the need to correctly answer membership queries directly influences the size of the test suite, both with respect to the number and the length of the test case execution traces. As the requirements of the membership queries lead to a larger test suite, they directly influence the complexity of the equivalence queries.

Proposition 1 (Complexity of the Equivalence Query on Finite Traces) *In the worst case, an equivalence query executes $|\mathrm{traces}(\mathcal{TS})|$ traces, where $|\mathrm{traces}(\mathcal{TS})|$ is the total number of test case execution traces in the test suite.*

Proof If the counter-example is the last trace in the test suite, the equivalence query executes all other traces before. ■

Corollary 1 (Execution of the Test Suite) *In the worst case, the learning algorithm needs $|S_{target}|$ executions of the test suite, where $|S_{target}|$ is the number of states in the target automaton.*

Proof This is a direct consequence of Proposition 1. In every iteration of the learning algorithm, exactly one equivalence query is made. The number of iterations in turn depends on the size of the resulting automaton, as in every iteration, exactly one new state is discovered. ■

While the complexity of the equivalence query (Proposition 1) suggests that a small test suite leads to a better performance of the equivalence query, it is nevertheless necessary that the test suite sufficiently covers the acceptable behavior of the SUT. In addition, as the closed world approach only accepts traces explicitly marked as *pass*, this implies the need for a large test suite. However, based on the structure of the membership queries, we can establish an upper bound on the required length of the test case execution traces.

Required Length of the Test Case Execution Traces

As we have elaborated in Section 4.1.2, our assumption of a closed world means that we have to represent every acceptable behavior so that membership queries can be answered correctly. For a system with cycles, this means that cycles have to be expanded for the test case execution traces. This amounts to path coverage of the SUT, and as explained in Section 3.4.2, the number and length of paths might be infinite. However, when taking a closer look at the queried sequences, an upper bound on the length of the test case execution traces can be established.

Theorem 2 (Required Trace Length) *The upper bound for the length of the test case execution traces is* $2 \cdot |S_{target}| + 1$*, where* $|S_{target}|$ *is the number of states in the target automaton.*

Proof The access strings used for identifying the states of the hypothesis automaton are derived from the counter-examples, i.e. the shortest prefix of the counter-example that identifies a new state is added to the classification tree. If we guarantee that always the shortest available counter-example is returned, then the length of the access strings is at most $|S_{\text{target}}|$, as the path from the start state to an arbitrary other state visits every state in the automaton at most once. The same upper bound holds for the length of the distinguishing sequences. Therefore, as the longest sequence to sift is derived as the concatenation of an access string w_a, a signal b from the signal alphabet, and a distinguishing string w_d at an internal node of the classification tree, $w_a b w_d$, the upper bound for the length of the test case execution traces is $2 \cdot |S_{\text{target}}| + 1$, one more than twice the number of states of the target automaton, so that membership queries can be answered in every iteration of the learning algorithm. ■

Though this limits the length of the test case execution traces, there are two drawbacks. First, in order to determine the shortest possible counter-example, the equivalence query has to process the test case execution traces against the hypothesis automaton in sequence to their length, which implies some kind of sorting of the test case execution traces. Second, in the general case, the number of states of the target automaton is unknown; therefore we cannot be sure whether the length of the traces is sufficient.

While the expansion of cycles is necessary to answer the membership queries, it also increases the number of test case execution traces in the test suite considerably, which in turn influences the complexity of the equivalence query. Ideally, we would like to be able to expand cycles as needed for the membership query, while keeping the test suite compact for the equivalence query. In consequence, we adapt the learning algorithm to infinite traces.

4.3 Learning from Infinite Traces

In Section 4.2.3, we stated that a central problem of the learning algorithm is the correct learning of cycles in the SUT, as we need traces of length up to the established upper bound to satisfy the expected membership queries (Theorem 2). The naive solution presented in Section 4.2, the unrolling of cycles up to a given length of the test case execution traces, has a number of drawbacks. First, the test suite gets larger, as for every additional execution of a cycle another test case execution trace is needed. Second, the test case execution traces get longer with the increasing number of cycle iterations. These two effects of the unrolling of cycles directly increase the size of the test suite, both in the number of test case execution traces and in the length of the test case execution cases. Therefore, both effects also influence the equivalence query, as in our definition of the equivalence, all test case execution traces are tried against the hypothesis automaton. In addition, there is another drawback of the unrolling of cycles, as the upper bound for the length of the test case execution traces depends on the number of states of the target automaton, which in the general case is not known beforehand.

In consequence, we need a more general way to represent cycles in the test case traces. The aim must be twofold. First, we need a representation of cycles that can be expanded on the fly, so that membership queries on repeated cycles can be answered without the need to know the maximum length of the queried traces beforehand. Second, we need a definition of the equivalence query which is more independent of the membership query's requirements on the size of the test suite.

4.3.1 Representing Cycles in the Test Cases

We already know that test cases themselves are software programs and show a control flow behavior similar to "normal" software. Particularly, test cases may also explicitly define cyclic behavior, e.g. iteratively polling a certain server. Until now, when generating the test case execution traces from a test case, we have generated a single trace for every possible path of the test case. In the case of a cycle, this meant a trace for every number of cycle iterations from zero up to a given upper bound on the total length of the trace.

In contrast, we now propose to mark cycles explicitly in the generated traces. Then, a cycle can be expanded on-the-fly when needed for a membership query, without needing to know the possible length of the queried trace beforehand. We denote a cycle as *(some signals)**, where signals inside the brackets are repeated in exactly the same order, but arbitrarily often. As the semantics of repeated cycle execution is identical to the semantics of the Kleene star, we choose the star, "*", as an easy to understand notation.

In Table 4.2, we show the test case execution traces for the coffee machine example (Figure 4.2 on Page 42) with explicit cycle representation. In comparison to the test case execution traces without explicit cycle information (Table 4.1 on Page 43), only about half the number of test case execution is needed. Also, for this example, we do not need to explicitly classify the empty trace as accepting, as this case is implicitly contained in the cyclic path, test case execution trace 1 in Table 4.2.

ID	Test Case Execution Trace	Verdict
1	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee)*	*pass*
2	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee,)* cm!requestCoffee	*fail*
3	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee,)* cm?outputCoffee	*fail*
4	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee,)* cm!insertMoney, cm!insertMoney	*fail*
5	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee,)* cm!insertMoney, cm?outputCoffee	*fail*
6	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee,)* cm!insertMoney, cm!requestCoffee, cm!insertMoney	*fail*
7	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee,)* cm!insertMoney, cm!requestCoffee, cm!requestCoffee	*fail*

Table 4.2: Test Case Execution Traces for the Coffee Machine With Explicitly Marked Cycles

4.3.2 Adaptation of the Teacher

As we represent cyclic behavior explicitly in the test case execution traces, we are now dealing with infinite traces. In consequence, we have to reconsider the query definitions from Section 4.2.1. There, a membership query was defined as the search for a test case execution trace, an accepting trace representing a member and a rejecting trace or a failed search representing a non-member of the concept to be learned (Definition 29 on

Page 30). This still holds for infinite traces, as the traces are unrolled on-the-fly when processing the test suite.

The equivalence query, however, needs to be redefined, as a complete execution of an infinite trace, i.e. iterated execution with increasing number of cycle iterations, is naturally impossible. The purpose of the equivalence is to prove that the hypothesis automaton conforms to all test cases in the test suite. This can be regarded as a structural test of the hypothesis automaton against the test suite. Therefore, instead of complete path coverage, we propose to execute boundary-interior path coverage of the test cases and limit the expansion of cycles to at most two. As in the formal definition of the equivalence query for finite test case execution traces, Definition 33, the first trace that does not reproduce its assigned verdict is returned as a counter-example.

4.3.3 Example

To show the advantages of infinite traces, we use the same example as in Section 4.2.2, a simple coffee machine (Figure 4.2). This time, we specify the test case traces with explicitly marked loops. The test case execution traces are listed in Table 4.2. The test suite contains less test case traces than the test suite in Table 4.1, as we can eliminate all explicit expansions of the loop.

While the smaller test suite allows for a more flexible membership query, the rest of the learning procedure works exactly as described in Section 4.2.2, even generating the same membership queries and the same counter-examples. We therefore only show the resulting target automaton (Figure 4.8).

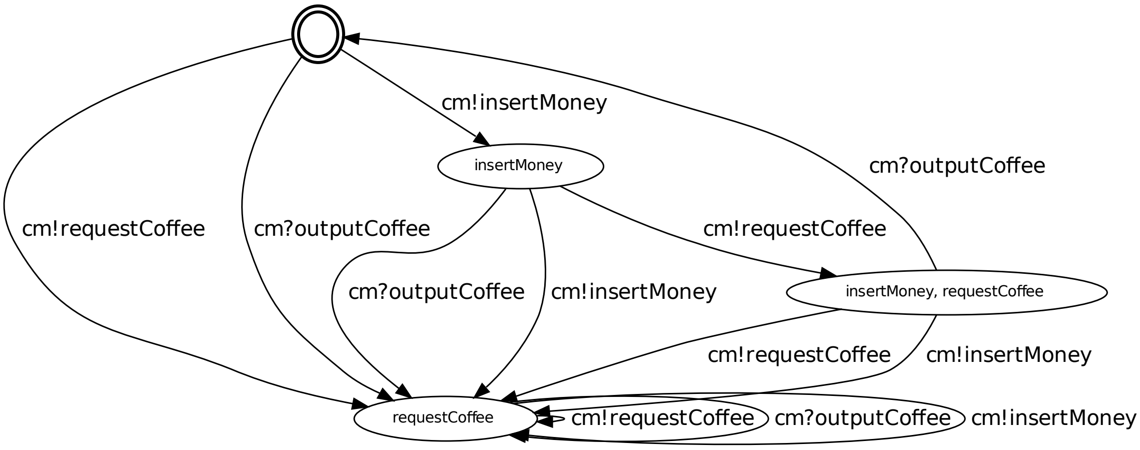


Figure 4.8: Automaton Learned from Infinite Traces

4.3.4 Decoupling Query Complexity and Test Suite Size

The query complexity of the membership query on finite traces (Theorem 1) still holds for infinite traces. The main advantage of the new approach is an increased independence of the queries' complexities, as the size of the test suite needed for the membership queries no longer directly affects the complexity of the equivalence query. Also, as cycles are expanded at need, the necessary length of the test case execution traces can be generated on-the-fly, without estimating the number of states in the target automaton beforehand.

Theorem 3 (Complexity of the Equivalence Query on Infinite Traces) *In the worst case, an equivalence query on infinite traces executes a boundary-interior coverage of the test suite.*

Proof As shown for the equivalence query on finite traces (Proposition 1), in the worst case, the counter-example is the last test case execution trace which is tried against the hypothesis automaton, which implies that the complete test suite has been executed before. If the test suite contains only finite traces, a boundary-interior coverage of the test suite amounts to executing every test case execution trace once against the hypothesis automaton, which is identical to the complexity of the equivalence query on finite traces (Proposition 1).

For an infinite trace, i.e. a trace that contains at least one cycle, it is necessary to establish a structural equivalence between the trace and the hypothesis automaton. This involves three steps. First, behavior specified in a cycle can be skipped. To check whether the hypothesis automaton correctly allows for skipping the according behavior, a trace without execution of the cycle is tried against the hypothesis automaton. Second, the behavior specified in a cycle can be executed. To check for the existence of the according behavior, a trace that executes the cycle once is tried against the hypothesis automaton. Third, the behavior specified in a cycle can be repeated. To check for the repeatability of the behavior, a trace that executes the cycle twice is tried against the hypothesis automaton. A further expansion of cycles is not necessary, as neither a threefold execution of a cycle nor the interleaved expansion of cycles can discover additional information on the structure of the automaton. The described manner of executing cycles, with zero, one, and two expansions of each cycle, corresponds exactly to the definition of a boundary-interior coverage on the test suite. ■

At a first glance, the complexity of the equivalence query on infinite traces seems to be worse than the complexity on finite traces. However, as stated in the proof for

Theorem 3, the difference between both complexities concerns solely the infinite traces. For infinite traces, it is necessary to execute a boundary-interior coverage of every cycle, amounting to three traces that have to be checked for each cycle. When learning from finite traces, traces specifying the expanded cycles are defined explicitly in the test suite, according to the required trace length (Theorem 2). Therefore, the traces that are generated by cycle expansion for the equivalence query on infinite traces are already contained in a test suite suitable for learning from finite traces. However, where the required length of the traces for learning from finite traces depends on the needs of the membership query, and thus on the number of states in the target automaton, the cycle expansions required by the equivalence query on infinite traces only depend on the cycles in the test case execution traces, and thus on the structure of the test suite.

In addition, a test suite for learning from infinite traces can be shorter than a test suite for learning from finite traces. Theorem 2 shows that the length of the required traces depends on the number of states in the target automaton. However, this length is chiefly needed for the correct detection of cycles. When cycles are marked in the test suite, they can be expanded on-the-fly for the membership query.

4.4 Summary: Adaptations of the minimally adequate teacher

In this chapter, we have described our adaptations of Angluin's learning algorithm to the conditions of test cases. We have analyzed the properties of the automaton generated by the learning algorithm and established that they are sufficiently adequate for our purposes. The main contribution of this chapter is therefore the definition of a *minimally adequate teacher (MAT)* that works on a test suite.

An MAT needs to answer two types of queries: the membership query and the equivalence query. We have defined the membership query as a search for a given trace on the test suite. If the search locates a trace that is assessed with the verdict *pass*, this trace is regarded as an accepting trace of the target automaton. In the same way, a queried trace that corresponds to a path in the test suite which is assessed with the verdict *fail* is regarded as a rejecting trace. Adhering to a closed world approach, we assume that every acceptable behavior is represented in the test suite and accordingly reject all traces for which no counterpart in the test suite can be found.

The equivalence query ensures that the learned automaton represents the data sample. Therefore, we have defined the equivalence query as a run of the test suite against the current hypothesis automaton, where the first test that fails to reproduce its verdict is returned as a counter example. We have argued that for establishing the equivalence of the hypothesis automaton to the test suite, it is sufficient to execute cyclic behavior in the test cases up to a maximum of two times.

While these definitions are sufficient for the learning procedure, we have observed that we need rather large test suites to be able to answer all membership queries correctly. In consequence, the efficiency of the queries, especially the membership query, depends on an efficient search strategy on the test suite.

5 Organizing the Sample Space using State Merging Techniques

In Chapter 4, we have defined and analyzed the formal requirements for the reconstruction of automata from test cases by learning. We have shown that for the equivalence query the traces from a test suite are sufficient, as long as the test suite satisfies a boundary-interior coverage of the *system under test (SUT)*. This also agrees with the results of Berg et al. [BGJ^{+}05]. However, it is still necessary to answer the membership queries correctly. As stated in Definition 32, we only accept traces that are labeled *pass*. This results in an automaton conforming very closely to the test suite. The drawback of this restrictive policy is that a falsely rejected trace, especially when updating the classification tree, can introduce errors into the state space that cannot be corrected. On the other side, in order to answer all membership queries correctly, we need a large number of test case execution traces, which leads to a large test suite. As a membership query essentially amounts to a search on the test suite, a large test suite also means increased effort in the query mechanism. Therefore, we need a representation of the test case execution traces which can be efficiently searched.

The method of automata inference by state-merging as introduced in Section 3.6 proposes a possible solution to both problems. In this method, a tree structure, the prefix tree acceptor, is used to organize the samples. On this tree, merging operations are performed, thus leading to a simpler and more general structure. While a tree structure can efficiently be searched for traces, the merging approach could be used to generate more traces for the membership queries. The drawback of this method is that the merging may lead to over-generalization, i.e. the constructed automaton would accept too many traces. Also, the method is based on a structural analysis of the prefix tree acceptor, which in our case would provide only marginal additional information as test cases are usually written to overlap as little as possible.

We therefore propose a different approach to state-merging. In addition to the structural information of the test cases we apply our knowledge of test case generation, thereby exploiting semantic properties to improve the structure. The advantage of our approach is a better preservation of the test suite structure, and it also identifies more additional information than the original state-merging method. Following the closed world philosophy, the additional traces generated in the trace graph are used twofold.

- Additional positive traces enhance membership queries, to avoid over-fitting.
- Additional negative traces mainly influence equivalence queries, where they may provide additional and possibly shorter counter-examples.

In the rest of this chapter, we will elaborate our approach. We start by defining a data structure for representing our training set, which consists of positive and negative test case execution traces. The trace graph is based on the prefix tree acceptor used in the state-merging method (Section 3.6). To be able to exploit the properties of the test cases best, the trace graph is modeled to closely represent our test cases. Subsequently, we describe the construction of the trace graph from the test cases and introduce the principles of semantic state-merging. Lastly, we relate the trace graph to the learning procedure we defined in Chapter 4.

5.1 Defining a Data Structure for State Merging

As described in Definition 27, the prefix tree acceptor used in the state-merging algorithm is defined as an automaton whose states are marked with prefixes of words from the language of the automaton to be learned. Since the prefix tree acceptor is merged into the final automaton, it contains only positive examples of the language and their prefixes. The start state of the prefix tree acceptor is marked with the empty word ϵ and the end states are labeled with the complete words from the positive sample. The length of the prefixes stored at the nodes increases with the distance to the root node, as in fact the prefix stored at a node describes the trace from the root node to that node. With respect to our application, two aspects of the prefix tree acceptor are most interesting. Every prefix of the positive sample, and thus the traces of the target automaton, is contained in the prefix tree automaton such that it begins in the start state. As traces may share a common prefix, they will also share a common prefix path in the prefix tree acceptor, thereby constituting a compact representation of the sample even before the actual state-merging.

For our notion of semantic state-merging, we also want to start with a compact representation of the sample. However, in contrast to the original state-merging approach, we do not want to generate the target automaton entirely from the prefix tree acceptor.

Instead, our goal is to define a structure that can be used as a basis to Angluin's learning approach. Therefore, we would like to represent our complete input data and in consequence, we need to represent positive and negative traces. Essentially, this means that we have to include the assessment of the traces in the data structure.

As our state-merging techniques are based on semantic knowledge about test case generation, we need a data structure which is closely related to the test cases themselves. In the end, our data structure shall be used to represent the complete test suite as input to the learning algorithm. In the following, we propose a search tree, called the *trace tree*, whose edges are labeled with the signals from the test cases and whose nodes are labeled with the verdicts from the test cases. In a second step, this data structure is refined to contain cycles.

5.1.1 The Trace Tree: Representing Test Cases

As described in Section 3.4.1, in general, a test case is itself a piece of software and can therefore be represented as a control flow graph. Usually, a test case distinguishes signals received from the SUT, signals sent to the SUT, and internal actions like the computation of some values or the setting of verdicts. As we have already explained in Section 4.1.2, we regard sent and received signals as input symbols to our target automaton and ignore internal actions except for the setting of verdicts. In consequence, we define the input alphabet of our test case automaton in Definition 34 as $\Sigma \cup V$, the set of sent and received signals Σ and the set of verdicts V. During the execution of a test case, the verdict may be changed at different points. The overall assessment of a test case depends on the verdicts set along the execution trace, which depends on the test language. We define the overall verdict of a test case execution path in the test case automaton by the function $v(w)$, which computes a verdict $\in V$ for a word w, $w \in (\Sigma \cup V)^*$ according to the rules of the test language.

Definition 34 (Test Case Automaton) A *test case automaton* is a *deterministic finite automaton (DFA)* $\mathcal{A}_{\mathrm{TC}} = (S, \Sigma \cup V, \delta, s_0, F)$, where S is a finite set of states, Σ a finite alphabet of signals, V is the set of verdicts, $\delta : S \times (V \cup \Sigma) \to S$ the transition relation, $s_0 \in S$ the start state, and $F \subseteq S$ a set of final states.

A transition in the automaton is labeled either by a signal or by a verdict, but not both at the same time. □

We call the verdict assigned at a given point during the test case execution *local verdict* as opposed to the *global verdict*, which is the verdict assigned to a complete test case execution trace. The global verdict can be computed from the local verdicts in the path, according to the rules of the used test specification language.

In general, every test case automaton combines a number of traces. At the same time, a test suite contains a number of test cases, where different test cases may contain identical traces as they partly overlap. In our data structure, we need to represent all traces from all test cases in the test suite, thereby eliminating duplicates and exploiting overlaps.

The *trace tree* is essentially a labeled search tree which represents all traces from a test suite. As in the prefix tree acceptor, all traces share the same starting state. This also corresponds to the learning algorithm's requirement that all input traces have to start in the same state. Also, as in the prefix tree acceptor, traces with common prefixes share their path in the trace tree as long as their prefixes match. In difference to the learning procedure, which only considers positive and negative traces, the trace tree models the test suite and therefore can contain all verdicts used by the test language. All in all, the trace tree can be seen as a union of all test cases.

Definition 35 (Trace Tree) A *trace tree* is a tree $(N, E, V, \Sigma, r, l, v)$, where N is a set of nodes, E is a set of edges, V is the set of verdicts, Σ is the signal alphabet, and r is the root node. The edges $e \in E$ are labeled with symbols from the signal alphabet Σ of the test suite by the edge labeling function $l : E \to \Sigma$. The nodes $n \in N$ are labeled according to the verdicts of the test case by the verdict function $v : N \to V$, which assigns a label to each node. The root node r corresponds to the start state of the SUT. Every path of the trace tree, starting with the root node and ending in a leaf of the tree, represents a trace of a test case in the test suite. □

Whereas in the test cases, the setting of a verdict is an internal action and thus part of the input alphabet of the automaton, in the trace tree, the verdict information is stored at the nodes. In this way, we establish a separation between information about the behavior of the SUT, i.e. signals sent to and received from the SUT, which is stored at the edges of the trace tree, and information about the assessment of the behavior of the SUT, like verdicts, which we store at the nodes of the trace tree.

To give an example of a trace tree, the finite test cases used for the coffee machine example in Section 4.2.2, listed in Table 4.1, are represented in the trace tree shown in Figure 5.1. For easier comparison, we repeat the test cases in Table 5.1.

Instead of storing prefixes at the nodes as in the prefix tree acceptor, we store single signals at the edges of our trace tree, as the structure then is easier to search. From the edge labels, we can compute the prefix $p(n)$ at a given node n of the trace tree as

ID	Test Case Execution Trace	Verdict
1	ϵ	*pass*
2	cm!insertMoney, cm!requestCoffee, cm?outputCoffee	*pass*
3	cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm!insertMoney, cm!requestCoffee, cm?outputCoffee	*pass*
4	cm!insertMoney, cm!insertMoney	*fail*
5	cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm!insertMoney, cm!insertMoney	*fail*
6	cm!insertMoney, cm?outputCoffee	*fail*
7	cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm!insertMoney, cm?outputCoffee	*fail*
8	cm!insertMoney, cm!requestCoffee, cm!insertMoney	*fail*
9	cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm!insertMoney, cm!requestCoffee, cm!insertMoney	*fail*
10	cm!insertMoney, cm!requestCoffee, cm!requestCoffee	*fail*
11	cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm!insertMoney, cm!requestCoffee, cm!requestCoffee	*fail*
12	cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm!requestCoffee	*fail*
13	cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm!requestCoffee	*fail*
14	cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm?outputCoffee	*fail*
15	cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm?outputCoffee	*fail*

Table 5.1: Test Case Execution Traces

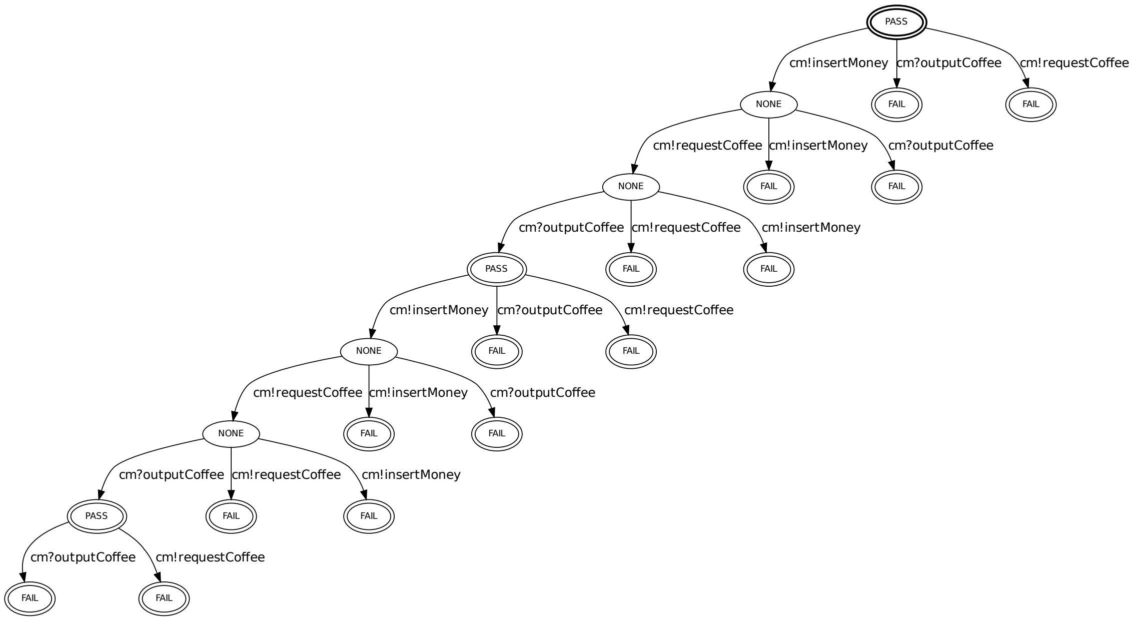


Figure 5.1: Trace Tree

follows. The start state n_0 is labeled with the empty word ϵ, $p(n_0) = \epsilon$. Then, each node is labeled as $p(n) = p(n_{\text{pre}})a$, where n_{pre} is the direct predecessor of n and a is the label at the edge from n_{pre} to n. The prefixes thus computed conform to the node labeling in the prefix tree acceptor as described in Definition 27. By using the prefix generating function $p(n)$ to label the nodes and only keeping traces in the tree that were rated as $v(w) =$ *pass*, the trace tree could be transformed into a prefix tree acceptor. However, for our purposes, it serves better to keep all traces in the same data structure.

The trace tree forms the basic data structure for our semantic state-merging. As the semantic state-merging methods depend on the information contained in the test cases, which in turn depends on the test language used in specifying the test cases, the trace tree can be extended to represent diverse structural information on the test cases by defining additional node labeling functions. As we already introduced explicit cycles into our learning procedure, we also need to represent cycles in the trace tree.

5.1.2 The Trace Graph: Including Cycles

The trace tree (Definition 35) adequately represents linear test traces. However, in Section 4.3, we have introduced explicit cycles into our *minimally adequate teacher (MAT)* and explained the advantages. To include cycles into the trace tree, we define an adapted structure, the *trace graph*. Cycles are represented by routing the closing edges back to the starting node of the cycle, therefore the structure is no longer a true tree. For better control, nodes where a cycle starts are marked as such.

Definition 36 (Trace Graph) A *trace graph* is a directed graph $(N, E, V, \Sigma, F, \delta, n_0, v, c)$, where N is a set of nodes, E is a set of edges, V is the set of verdicts, Σ is the signal alphabet, $F \subseteq N$ is the set of final nodes, $\delta : N \times \Sigma \to N$ is the transition relation, n_0 is the start node, $v : N \to V$ is the verdict function that marks each node in the trace graph with the according verdict, and $c : N \to \{\text{true}, \text{false}\}$ is a node labeling function that marks nodes where a cycle starts. The root node n_0 corresponds to the start state of the SUT. A path through the trace graph beginning in the start node and ending in an final node corresponds to a trace of a test case in the test suite. □

With the trace graph structure, cycles in the SUT still need to be defined explicitly in the test suite, but they are represented in a more flexible way for the learning algorithm. Incidentally, the trace graph itself will be smaller in size compared to the trace tree, as now only one cycle will be stored where before a number of unrolled iterations of the same cycle had to be included.

As an example, we show the trace graph for the infinite test cases from Section 4.3.3 in Figure 5.2. Again, for easier reference, we repeat the test case execution traces in Table 5.2. In comparison to the trace tree in Figure 5.1, the trace graph in Figure 5.2 is more compact, but contains an even larger number of traces.

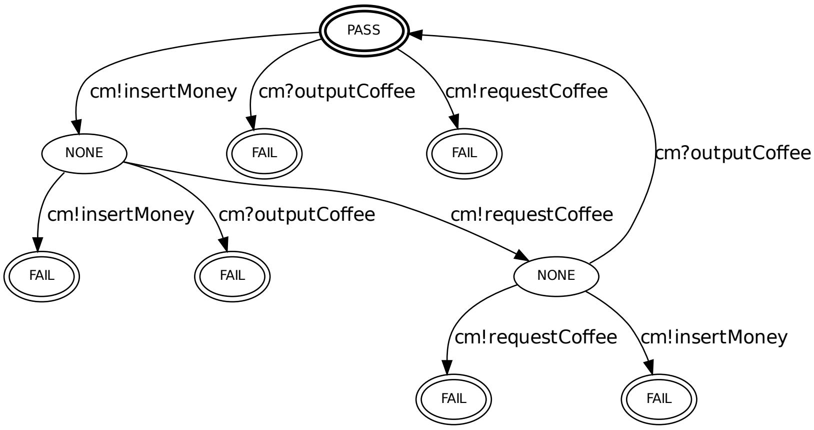


Figure 5.2: Trace Graph

While the adaptation to the representation of cycles is the most important one for our learning approach, additional structural information on test cases can easily be added

ID	Test Case Execution Trace	Verdict
1	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee)*	*pass*
2	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee)*, cm!requestCoffee	*fail*
3	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee)*, cm?outputCoffee	*fail*
4	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee)*, cm!insertMoney, cm!insertMoney	*fail*
5	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee)*, cm!insertMoney, cm?outputCoffee	*fail*
6	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee)*, cm!insertMoney, cm!requestCoffee, cm!insertMoney	*fail*
7	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee)*, cm!insertMoney, cm!requestCoffee, cm!requestCoffee	*fail*

Table 5.2: Test Case Execution Traces

by introducing extra node labeling functions. That way, information on the test cases will only affect the construction of the trace graph, but not the learning procedure that depends on its structure.

5.2 Constructing the Trace Graph from Test Cases

In Section 5.1, we have defined a general representation of test cases, the test case automaton (Definition 34). This representation applies to all test specification languages, and therefore the trace graph can be used to represent test cases of all test specification languages. When generating the trace graph, however, the properties of the used test specification language have to be considered, as they influence the compatibility of traces and thus our state-merging approach. While the overall approach to the trace graph and state-merging is universal, some details, like the handling of verdicts, are specifics to the semantic of the test specification language. Where this is the case, we adhere to the regulations of the test specification language *Testing and Test Control Notation (TTCN-3)* [ETS07].

In the following, we will show how the trace graph is constructed from the traces of the test cases in the test suite. We start with a universal algorithm for adding linear and cyclic traces. Subsequently, we highlight the influence of verdicts on the construction and introduce the first two state-merging techniques, to process cycles and default branches in the test cases.

5.2.1 Adding Traces

A test case automaton as defined in Definition 34 combines a number of test case execution traces, each associated with a control flow path. In most cases, there is a test purpose which corresponds to a *pass* trace and constitutes the main flow of the test case as well as a number of deviations corresponding to erroneous reactions of the SUT which result in the verdict *fail*. This way, the same (partial) behavior can be referred to by different test cases, being the focus of the test in one test case and a sideline in another. As the trace graph is essentially a prefix tree acceptor, common prefixes of test case traces are mapped onto the same graph nodes. To construct the trace graph, we separate the test cases into single traces and add them to the trace graph.

Figure 5.3 shows an example test case automaton, Table 5.3 the associated traces. In the test case automaton, the verdicts are represented as internal signals and therefore associated with edges of the test case automaton.

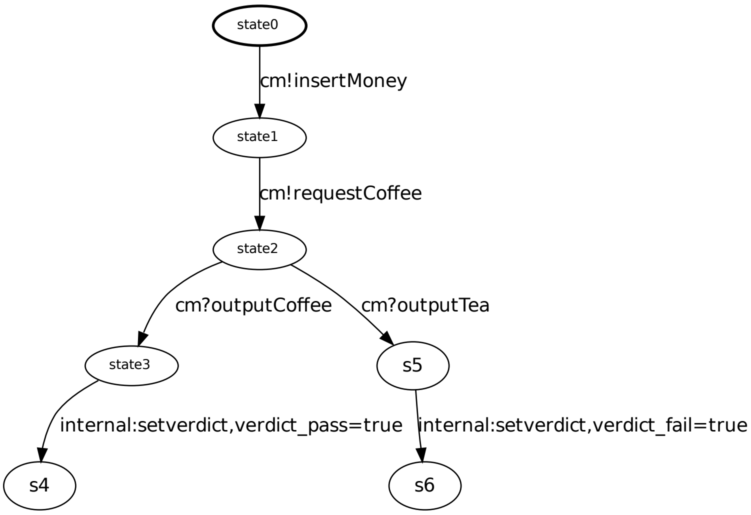


Figure 5.3: A Test Case Automaton

In the trace graph, verdicts are associated with the nodes of the graph, whereas they are associated with special edges in the test case automaton. Therefore, when adding a test case trace to a trace graph, the verdicts move from an edge to a node. To this end, when parsing the test case automaton, we check for the keyword indicating the setting of a verdict. In the test language TTCN-3, for example, this keyword is the function

ID	Test Case Execution Trace	Verdict
1	cm!insertMoney, cm!requestCoffee, cm?outputCoffee	*pass*
2	cm!insertMoney, cm!requestCoffee, cm?outputTea	*fail*

Table 5.3: Traces of the Test Case Automaton

setverdict. On reading this keyword, instead of adding a new edge the current node is labeled with the appropriate verdict.

When adding a linear trace to the trace graph, we start in the start node of the trace graph and look at the first signal in the trace. If the current node already has an edge labeled with the current signal, we proceed along this edge to its target node in the trace graph and proceed to the next signal in the trace. This is repeated until we reach a node where none of the outgoing edges matches the current signal. Then, the current suffix of the trace is added as a new subgraph of the current node. Algorithm 5 shows the algorithm in pseudo code.

```
Data: A trace $w$
Start at the root node $n_0$ of the trace graph;
for all signals in $w$ do
    Get the first signal $b$ in $w$;
    if the current node has an outgoing edge marked $b$ then
        Move to the $b$-successor of $n$, which is $\delta(n, b)$;
        Remove the first signal from $w$;
    else
        // The signal is unknown at the current node
        Add $w$ as a new subgraph at the current node;
        return;
    end
end
```

Algorithm 5: Add a Trace to the Trace Graph

5.2.2 Adding Cycles

Cycles of the test case automaton need to be specially treated, as a cycle means that an edge loops back to an existing node. A trace containing a cycle can be separated into three parts $w = w_{\text{pre}} w_{\text{cycle}} w_{\text{post}}$, where w_{pre} is the sequence of signals leading into the cycle, w_{cycle} is the cycle itself, and w_{post} is the sequence of signals leading out of the cycle.

The strings w_{pre} and w_{post} may also be empty. When adding a trace containing a cycle, we proceed in three steps, adding the linear parts of the trace as described above and closing the cycle explicitly. First, we add w_{pre} and remember the last node n_{cycle}, which marks the beginning and ending of the cycle. In the next step, we split the cycle into a prefix and the last signal, $w_{\text{cycle}} = w_{\text{cycle}-1}a$. The prefix of the cycle, $w_{\text{cycle}-1}$ is a linear sequence and is accordingly added. Again we remember the last node n_{last} and add the closing edge of the cycle as $(n_{\text{last}} \xrightarrow{a} n_{\text{cycle}})$. In the last step, we proceed by adding w_{post} starting from n_{cycle}. The pseudo code is given in Algorithm 6.

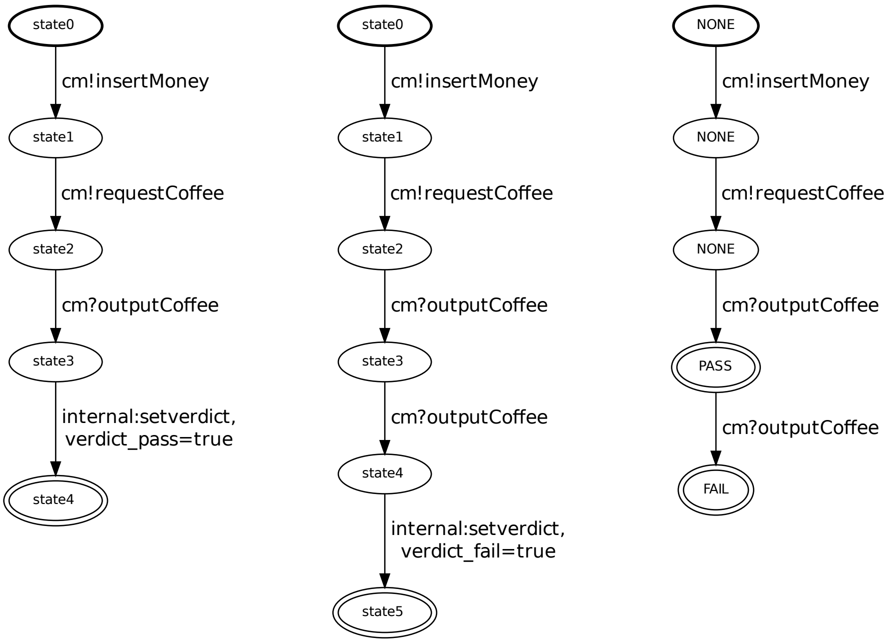
```
Data: A trace $w$
Partition the trace: $w = w_{\text{pre}} w_{\text{cycle}} w_{\text{post}}$;
Add $w_{\text{pre}}$ to the trace graph;
Let $n_0$ the root node of the trace graph;
Find the target node $n_{\text{cycle}}$ on the trace graph as $n_{\text{cycle}} = \hat{\delta}(n_0, w_{\text{pre}})$;
Let $a$ the last element of $w_{\text{cycle}}$, which is $w_{\text{cycle}}[|w_{\text{cycle}}|]$;
Remove the last element $w_{\text{cycle}}[|w_{\text{cycle}}|]$ from $w_{\text{cycle}}$;
Add $w_{\text{cycle}}$ to the trace graph, starting at node $n_{\text{cycle}}$;
Find the target node $n_{\text{cycle}-1}$ on the trace graph as $n_{\text{cycle}-1} = \hat{\delta}(n_{\text{cycle}}, w_{\text{cycle}})$;
Add a transition from $n_{\text{cycle}-1}$ to $n_{\text{cycle}}$ so that $\delta(n_{\text{cycle}-1}, a) = n_{\text{cycle}}$;
Add $w_{\text{post}}$ to the trace graph, starting at node $n_{\text{cycle}}$;
```

Algorithm 6: Add a Cycle to the Trace Graph

5.2.3 Verdicts

While adding traces to the trace graph, we check whether the verdicts in the currently processed trace are compatible to the verdicts already in the trace graph. In Figure 5.4, an example for compatible verdicts is given in the terms of the coffee machine. Figure 5.4a shows an accepted sequence, where the request for a cup of coffee is followed by the output of the same. The corresponding test case execution trace is cm!insertMoney, cm!requestCoffee, cm?outputCoffee, *pass*. The sequence in the second test case, shown in Figure 5.4b, is only one signal longer than the sequence of the first test case—instead of stopping after outputting the coffee, the machine outputs another cup. Clearly, this sequence of events is rejected, as the output of the second coffee which was not requested could make the cup overflow. Accordingly, the test case execution traces of the second test case is cm!insertMoney, cm!requestCoffee, cm?outputCoffee, cm?outputCoffee, *fail*.

The trace graph storing the two test cases is shown in Figure 5.4c. It is obvious that both test case execution traces can be extracted correctly.



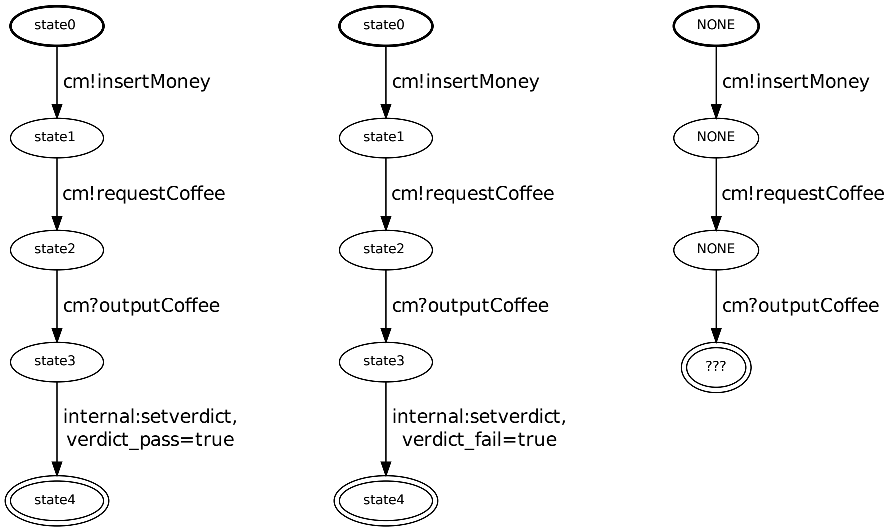
(a) Test Case 1 (b) Test Case 2 (c) Trace Graph

Figure 5.4: Compatible Verdicts in the Trace Graph

Whereas it is acceptable if the verdict is changed along a trace, it is not acceptable if two traces put different verdicts at the same node. An example is shown in Figure 5.5, where the test cases in Figures 5.5a and 5.5b share exactly the same sequence of signals, but apply a different verdict. In the construction of the trace graph in Figure 5.5c, it is therefore not clear which verdict to put into the final node.

In the example (Figure 5.5), the mismatch of the verdict is due to an error in the postamble. In general, finding incompatible verdicts in the same node of the trace graph can imply that the same sequence of signals—and therefore the same behavior—was rated in different ways in the same test suite. This may indicate inconsistencies or errors in the test suite. In any case, learning from such a test case is not recommendable, as there is no safe way of guessing which of the two competing verdicts is the correct one.

So far, the state-merging in the trace graph only means the combination of the test case automata, where traces are only merged as far as their prefixes match. The trace graph therefore exactly represents the test cases, but nothing more. In the following, we show two techniques to derive additional traces based on our knowledge of test cases.



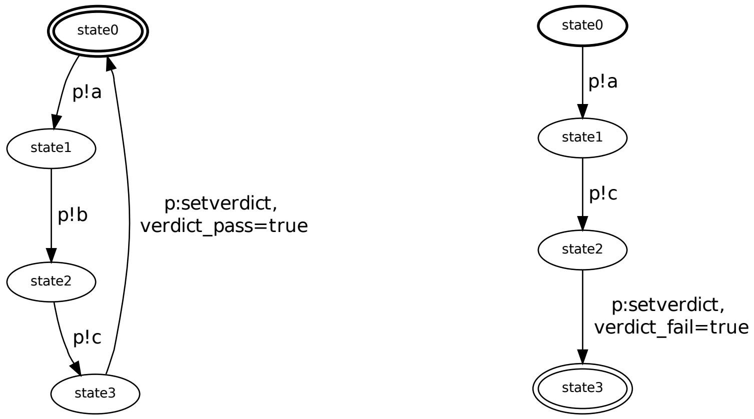
(a) Test Case 1 (b) Test Case 2 (c) Trace Graph

Figure 5.5: Incompatible Verdicts in the Trace Graph

5.2.4 Cycles and Non-Cycles

When testing a software system with repetitive behavior or a cyclic structure, the cycle has of course to be tested. However, it is usually sufficient to test the correct working of the cycle in one test case. In all other test cases the shortest possible path through the software is considered, which may mean that a test case executes only a part of a cycle or completely ignores a cycle. Depending on the test purpose, the existence of the cycle might not even be indicated in the test case. As long as the cycle itself is tested by another test case, the test coverage is not influenced. This approach results in shorter test cases, which usually means shorter execution time and thus faster testing. Also, readability of the test cases is increased. While the preselection of possible paths for cycles is appropriate for software testing, for machine learning it is desirable to have access to all possible paths of the software.

Consider the two test cases shown in Figure 5.6, which are part of the example in Section 4.3.3. Although this is only a small example for demonstration purposes, the setting is quite typical. The test case shown in Figure 5.6a tests the positive case, that is, a repeated iteration of the three signals a, b, and c. The test case shown in Figure 5.6b tests for a negative case, namely what happens if the system receives an inopportune signal. Here, the repetitive behavior is ignored, as it has been tested before and the test focus is on the error handling of the system. However, usually this behavior could also be observed at any other repetition of the cycle.



(a) A Test Case with a Cycle

(b) A Test Case without a Cycle

Figure 5.6: Test Cases with and without Cycles

For the learning procedure, we would like to have all those possible failing traces, not only the one specified. We therefore define a precedence for cycles, which means that whenever a cycle has the same sequence of signals as a non-cyclic trace, the non-cyclic trace is “folded” into the cycle. For the example test cases in Figure 5.6, the trace graph is shown in Figure 5.7. Besides the trace p!a, p!c, *fail* explicitly specified in the test case shown in Figure 5.6b, the trace graph also contains the traces where the cycle is executed, (p!a, p!b, p!c)*, p!a, p!c, *fail*.

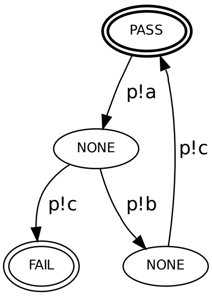


Figure 5.7: Trace Graph with Folded Trace

With precedence of cycles, the test suite used as input to the learning algorithm can be more intuitive, as cycles only need to be specified once. For the test suite of the coffee machine example (Section 4.3.3), this simplifies the test traces considerably. Table 5.4 shows the test cases from Section 4.3.3, where the cycle is specified in every test case. In

Table 5.5, the cycle is only specified in test case 1. The cyclic behavior of the other test cases is determined via state-merging on the trace graph.

ID	Test Case Execution Trace	Verdict
1	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee)*	*pass*
2	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee)*, cm!requestCoffee	*fail*
3	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee)*, cm?outputCoffee	*fail*
4	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee)*, cm!insertMoney, cm!insertMoney	*fail*
5	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee)*, cm!insertMoney, cm?outputCoffee	*fail*
6	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee)*, cm!insertMoney, cm!requestCoffee, cm!insertMoney	*fail*
7	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee)*, cm!insertMoney, cm!requestCoffee, cm!requestCoffee	*fail*

Table 5.4: Test Suite with Explicit Cycles

ID	Test Case Execution Trace	Verdict
1	(cm!insertMoney, cm!requestCoffee, cm?outputCoffee)*	*pass*
2	cm!requestCoffee	*fail*
3	cm?outputCoffee	*fail*
4	cm!insertMoney, cm!insertMoney	*fail*
5	cm!insertMoney, cm?outputCoffee	*fail*
6	cm!insertMoney, cm!requestCoffee, cm!insertMoney	*fail*
7	cm!insertMoney, cm!requestCoffee, cm!requestCoffee	*fail*

Table 5.5: Test Suite with Implicit Cycles

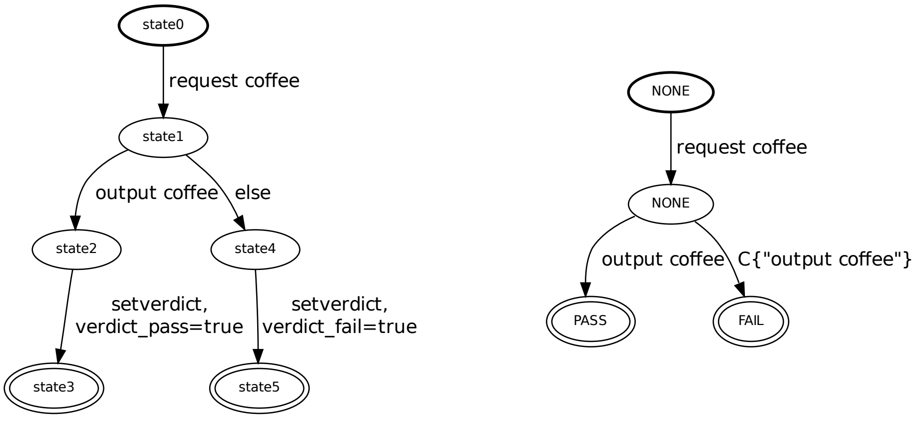
5.2.5 Default Behavior

Another common feature of test cases is the concentration on one test purpose. Usually, the main flow of the test purpose forms the test case, while unexpected reactions of the SUT are handled in a general, default way. Still, there may exist a test case that tests (a part of) this default behavior more explicitly.

Default branches usually occur when the focus of the test case is on some specified behavior, and all other possible inputs are ignored or classified as *fail*. Also, sometimes a test case only focuses on a part of the system, where not all possible signals are known. In such cases, the test case often contains a default branch, which classifies what is to be done on reading anything but what was specified.

For our application, this poses two challenges. The first challenge is for the learning procedure. For the different queries, we would like to have as many explicitly classified traces as possible, but at the same time we would not want to blow up the size of the traces too much. The second challenge is in the construction of the trace graph. When adding all different traces into one combined structure, the implicit context of what is "default" in the local test case is lost. Also, sometimes another test case uses the same default, adds more specific behavior in the range of the default, or defines a new default which slightly differs. We therefore need a method of preserving the local concept of "default" in the test cases and a method of combining different defaults in the trace graph.

Consider a typical default situation, like a **default** statement in a **case** environment. The **default** collects all cases that are not explicitly handled beforehand. As branching on alternatives splits the control flow in a program, each of the branches belongs to a different trace. Therefore, when taking the traces one by one, the context of the default is not clear. To preserve this context, instead of **default** we record the absolute complementary of the set of other alternatives, which is $\complement\{a, b\}$. Figure 5.8 shows a test case with defaults (Figure 5.8a) and its representation as a trace graph (Figure 5.8b).



(a) Test Case with Default

(b) Trace Graph

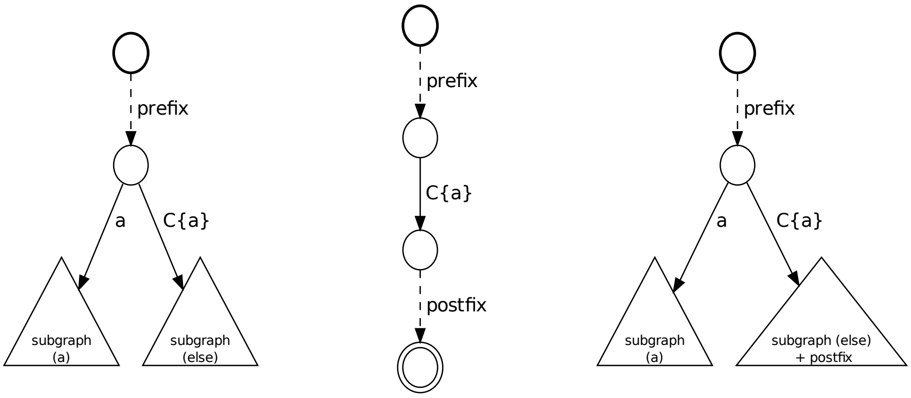
Figure 5.8: Representing Defaults in the Trace Graph

When adding traces to a trace graph containing defaults, three cases have to be distinguished.

- Add a trace with a matching complementary set.
- Add a trace that singles out one signal from the complementary set.
- Add a trace with a different complementary set.

Let us split the traces to consider in three parts, $t = pdq$, where t denotes the trace to be added, p denotes the common prefix up to the default statement, d denotes the default statement or the signal at the place of the default, and q denotes the postfix following the default statement.

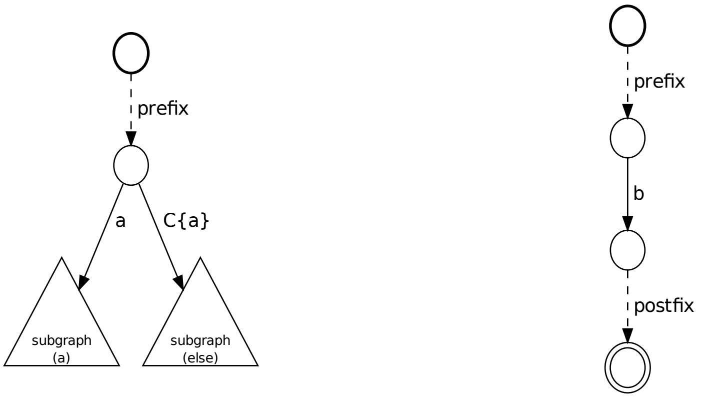
The first and simplest case is to add a trace with a matching default and thus a matching complementary set. As the complementary sets are identical, it suffices to add the postfix of the trace to the subgraph of the default already in the trace graph. Figure 5.9 illustrates this. Figure 5.9a shows the part of the trace graph containing the default before the new trace is added. As only the nodes surrounding the default edge are of interest, we show the rest of the trace graph in a stylized way. In Figure 5.9b, we show the trace to be added, and Figure 5.9c depicts the resulting trace graph after the new trace was added. The postfix from Figure 5.9b is integrated into the subgraph of the edge marked with the complementary set.



(a) Trace Graph with Default (b) Test Trace with Matching Default (c) New Trace Graph

Figure 5.9: Adding a Trace with Matching Default

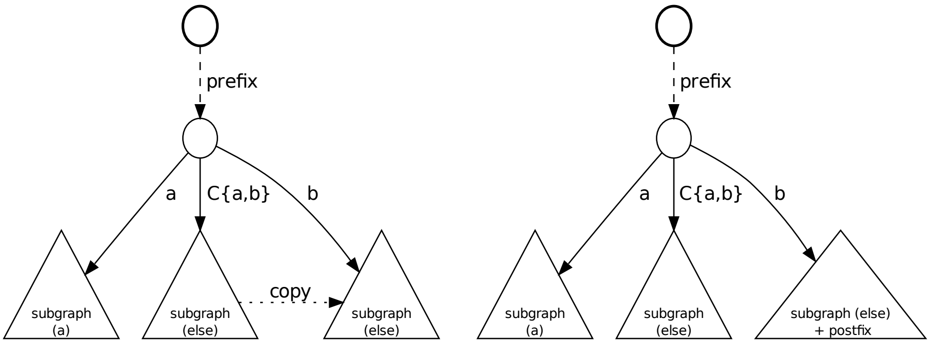
In the second case, we want to add a trace that shares the same prefix as the default, but does not match any of the existing alternatives. The situation is depicted in Figure 5.10, the signal following the prefix in the trace shown in Figure 5.10b is not exactly matching the existing branches of the trace graph shown in Figure 5.10a. However, the trace would match the default branch, only not the whole complementary set but a single item thereof. Therefore, we split the default branch as shown in Figure 5.11a. The subgraph of the default branch is copied, so that the new branch has the same postfixes as before. Then, we can add the postfix of the new trace to the subgraph of the new branch as described for the simple case. This is shown in Figure 5.11.



(a) Trace Graph with Default

(b) Test Trace with Additional Alternative

Figure 5.10: Adding a Trace with an Additional Alternative: Initial Situation

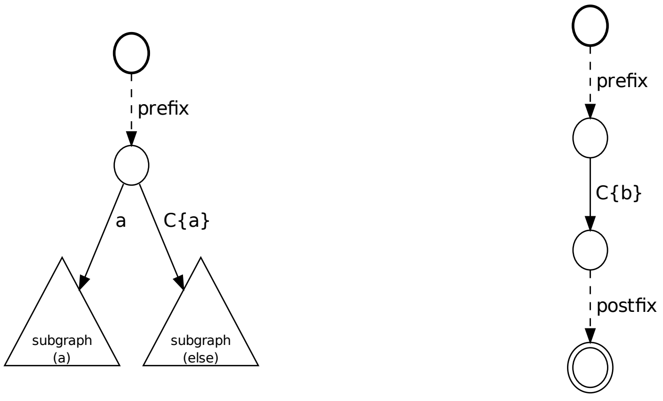


(a) Splitting the Default Branch

(b) New Trace Graph

Figure 5.11: Adding a Trace with an Additional Alternative: Adding to the Split Default Branch

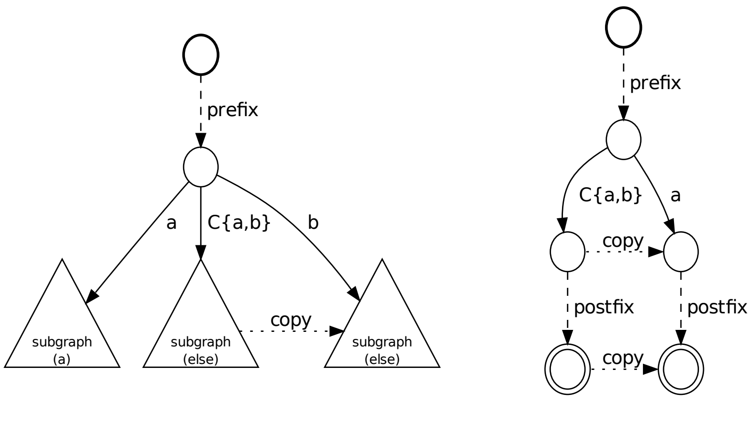
In the third and last case, we add a trace with a different default statement to a trace graph. The situation is depicted in Figure 5.12. The trace graph contains an edge marked with the complementary set $\mathtt{C}\{a\}$ and the test trace contains an edge marked with the complementary set $\mathtt{C}\{b\}$ The problem, as well as the solution, is similar to that of the second case, but now it is bidirectional. The complementary set of the test trace to be added does not fit the complementary set of the trace graph. As before, we split the default branch of the trace graph, such that the edge marked b is branched out from the complementary set (Figure 5.13a). The remaining complementary set in the trace graph is $\mathtt{C}\{a, b\}$. However, the complementary set of the test trace still does not match, therefore we split the test trace, such that the edge marked a is branched out from the complementary set (Figure 5.13b). The complementary sets of the trace graph and the test trace are now identical, $\mathtt{C}\{a, b\}$, but the test trace has been split into two test traces. Then we add the two resulting test traces traces, resulting in the trace graph shown in Figure 5.14.



(a) Trace Graph with Default

(b) Test Trace with Additional Alternative

Figure 5.12: Adding a Trace with an Additional Default: Initial Situation



(a) Splitting the Default Branch in the Trace Graph

(b) Splitting the Default Branch in the Test Trace

Figure 5.13: Splitting Default Branches

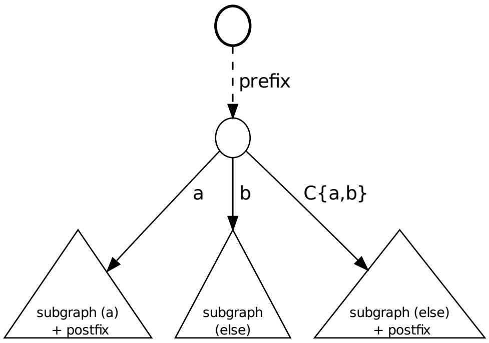


Figure 5.14: New Trace Tree

The described technique also generalizes to sets with more than one element. In this case, the sets associated with the split branches are determined as the intersections and differences of the given sets. An example is shown in Figure 5.15, where A is the complementary set of the default to split and B is the complementary set of the default in the test trace. To put it simple, a common partition of the sets is built, both for the trace graph and for the test trace, so that traces with exactly matching sets can be added.

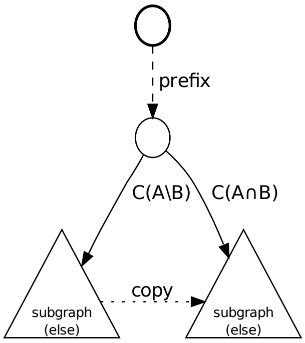


Figure 5.15: Splitting Default Branches in the General Case

5.3 Learning from the Trace Graph

We have defined the trace graph to represent test cases and to be able to reflect test language specific details. For the learning process, we need to abstract from those details. In Chapter 4, we have described test cases as single traces that end in a verdict. Based on that, we defined the notion of test case execution traces, which couple a trace of input and output signals of the SUT with a verdict assessing the whole trace. This description is most suitable for the learning algorithm, as the learning algorithm expects a trace and a classification of the complete trace. In the trace graph, verdicts are stored locally and may change along a path through the trace graph. This representation allows to store traces of different length and with different, yet compatible, verdicts in the same graph. For learning from the trace graph, we need to map the local verdicts stored in the trace graph to the global verdicts the learning algorithm expects. Besides the computation of the global verdict of a trace, we need to relate the learning procedure's querying mechanisms to the trace graph. As the trace graph is designed to be efficiently searchable, we will see that the membership queries benefit from our data representation.

5.3.1 Computing the Verdicts

When dealing with the verdicts in the trace graph, there are two issues to solve. First, we need to ascertain the compatibility of the local verdicts, as incompatible verdicts will cause problems in the learning procedure. Second, we need a method to compute the global verdict from the local verdicts along a path of the trace graph.

The first problem is already solved on constructing the trace graph (Section 5.2.3). When a mismatch of local verdicts is detected, the learning procedure is interrupted, as it is impossible for the learning procedure to decide which verdict should be correct.

The second question has to be solved according to the semantics of the test specification language. In TTCN-3, the policy is that a verdict can only get worse. In consequence, the global verdict of a trace is the worst verdict along the path in the trace tree. As the learning procedure only accepts *pass* and *fail*, complemented with the verdict *none* that indicates that no verdict has been set, the global verdict of a trace can be computed according to Definition 37.

Definition 37 (Global Verdict of a Trace) Let w be a trace in the trace graph $\mathcal{G} = (N, E, V, \Sigma, F, \delta, n_0, v, c)$ starting at the root node n_0 of the trace graph. The set of local verdicts $V(w)$ of a test trace w is the collection of all local verdicts v_l stored at the target nodes of all prefixes of w,

$$V(w) = \{v_l \mid \exists w_p \in \mathrm{Pr}(w) : v_l = v(\hat{\delta}(n_0, w_p))\} .$$

Then, the global verdict v_g of the trace w is computed as

$$v_g(w) = \begin{cases} \textit{fail} & \forall v_l \in V(w) : v_l = \textit{none} \\ \textit{fail} & \exists v_l \in V(w) : v_l = \textit{fail} \\ \textit{pass} & \text{otherwise} . \end{cases}$$

In other words, for a trace w to be accepted, there must be at least one *pass* verdict and no *fail* verdicts among the local verdicts. □

5.3.2 Queries on the Trace Tree

The trace graph is designed to resemble a search tree, where the successor of a state is reached by the next signal in the sequence to process. Therefore, a membership query on the trace graph only depends on the length of the queried sequence, which in turn depends on the length of the access strings and distinguishing strings and therefore on the length of the counter-example. As explained in Section 4.1.2, the assessment of a trace that is not found in the test suite depends on the adoption of a closed world or open world approach. In congruence with the membership query defined in Definition 28, we rate a trace that is not found on the trace graph as not acceptable and therefore apply the verdict *fail*. We apply the same policy to incomplete traces, i.e. the queried trace doesn't end in a final state or no verdict has been applied during the trace.

In the equivalence query, we need to test every complete trace of the test suite against the hypothesis automaton. This means that we have to extract every trace from the trace graph. As we already explained in Section 4.3.4, it is enough to expand cycles twice for the equivalence query. Therefore, a tree walking algorithm can be applied to the trace graph, as long as the number of cycle expansions is registered and the generated traces are recorded to keep track of interleaved cycles. Yet, as we still prefer short counter-examples, a simple depth-first tree walking algorithm is not sufficient. To extract the shortest possible counter-example in each iteration, we need to use an iterative deepening search. The complexity of such an iterative deepening search depends on the branching factor of the tree to be searched. For the trace graph, the branching depends on the structure of the test suite. An upper bound on the branching factor is the number of signals of the SUT. However, usually not every signal is explicitly tested against every state of the SUT, therefore in most cases the branching of the trace graph would be limited to a small number of signals and a default branch.

5.4 Summary: A Generic Data Structure for Learning

We have introduced the trace graph, a data structure based on a search tree, to represent all traces of a test suite in a single graph. As the prefix tree acceptor of the state-merging approach, the trace graph maps common prefixes of the test case traces onto the same path in the trace graph. Based on this property, we use the characteristics of test cases and the trace graph to generate additional traces for the learning procedure. We call our proposed state-merging techniques *semantic* and *conservative*, as we use semantic knowledge about the construction of test suites rather than structural information of the traces and only merge traces where it is justified by the properties of the test cases. Thus, we only generate traces in the trace graph that comply with the test suite.

The trace graph is constructed from the test cases by separating the test cases into traces and adding these traces one by one to the trace graph. Based on semantic properties of test cases, the traces of the test suite are merged during the construction of the trace graph. We have described two basic notions for the merging of test traces, cycles and defaults. While these are present in most test specification languages, there may well be other properties that can be used in generating a trace graph, such as global testing states.

Traces that share part of a cycle trace are considered as implicitly cyclic. In consequence, they share the cycle nodes in the trace graph. The precedence of cycles allows the test suite used for learning to be smaller and more intuitive.

Default branches in the test cases simplify the trace graph by fusing many branches into one, but when different default branches share a common prefix, it has to be ensured that the subsequent behavior is correctly attached. To this end, we have defined default branches via their complementary set, and described the merging of branches depending on the relation of their complementary sets, i.e. equal sets, different sets, and overlapping sets.

We have also explained how the queries are processed on the trace graph and shown that the trace graph greatly enhances the membership query, where the complexity of a single membership query now only depends on the length of the queried trace instead of on the complexity of a search on the test suite. The trace graph is also suitable for the execution of an equivalence query. As we always want to generate the shortest counterexample possible, we need to perform an iterative deepening search. This also is easier to establish on a centralized graph structure than on separate test cases.

The trace graph also separates the opposing interests of analyzing the test specification language for an extensive exploitation of the test suite and a universal approach to learning from test cases. With the trace graph, the semantic state-merging can be used to generate traces according to the needs of the test specification language, and to store the traces in a general way that can be used by the learning algorithm.

6 Implementation and Case Study

To assess the power of our learning approach, we have developed a prototypical implementation [AOWZ09, Ott09]. The implementation realizes an Angluin-style learner, including the adaptations to the queries described in Chapter 4 and the organization of the test data into a trace graph as discussed in Chapter 5. Using the prototype, we perform a case study based on a well-known problem, the *conference protocol* [BRS$^+$00]. The conference protocol describes a chat-box program that can exchange messages with several other chat-boxes over a network.

The aim of our case study is threefold. First of all, we want to validate our adaptations on the query mechanisms of Angluin's learning algorithm and our notion of semantic state-merging. To this end, we reconstruct a known model from its test cases. The second goal of our case study is to assess the need for completeness of the test suite. In Section 4.1.2, we observe that to learn the correct automaton, the membership queries have to be answered correctly, which in the end suggests a test suite realizing a path coverage of the *system under test (SUT)*. Therefore, we try test suites satisfying different coverage criteria, to assess the impact of the test suite's coverage on the success of the learning process. Thirdly, we want to draw conclusions on if and how the structure of the SUT influences the learning process from the experiments.

For our experiments, we consider two versions of the conference protocol and generate two sets of test cases for each of them. In the first version of the conference protocol, we restrict the valid signal sequences to limit the number of paths of the SUT. The second version of the conference protocol includes a wider range of signal sequences. For both versions of the conference protocol, we first outline the expected structure of the model to be learned, and then generate increasingly complex test suites until the learned automaton matches the expected structure.

In the following, we will give a short overview of the prototypical implementation and describe our setting of the conference protocol. Subsequently, we describe the results obtained when learning from the different sets of test traces. In the last section of this chapter, we will compare the two experiments and draw some conclusions.

6.1 Prototypical Implementation

Our prototype is implemented in the programming language Java, the abstract structure is shown in Figure 6.1. The class `Learner` implements Angluin's learning algorithm. In every iteration of the learning algorithm, a new counter example is obtained via an equivalence query and used to detect a new state. The discovered states are organized in a classification tree, which is also used to generate the new hypothesis automaton as described in Section 3.7. The two queries, equivalence query and membership query, are implemented according to our adaptation to learning from test cases (Chapter 4), and mapped onto a trace graph structure as defined in Chapter 5. The prototype implements the equivalence query for infinite traces (Section 4.3.1). However, the equivalence query for finite traces can be emulated via the structure of the input test suite: for traces without cycles, the behavior of the equivalence query is the same for finite and infinite traces.

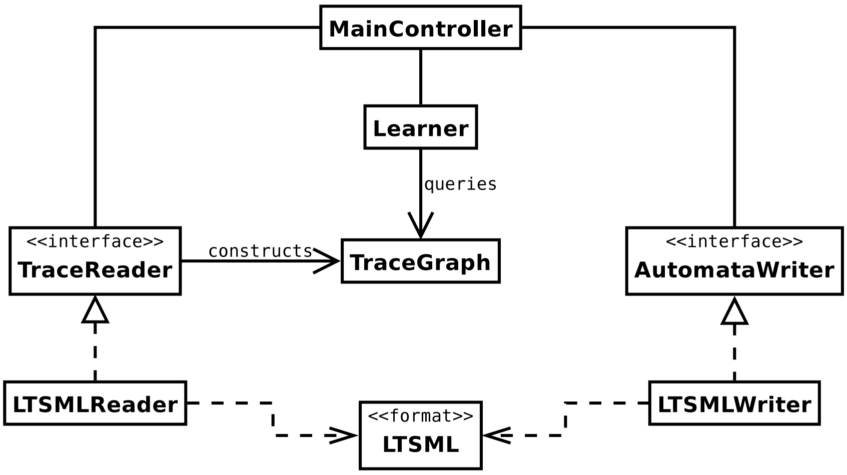


Figure 6.1: Abstract Structure of the Implementation

In the class `TraceGraph`, the basic methods of semantic state-merging are implemented. A trace graph structure is constructed by adding traces from test cases. The precedence of loops (Section 5.2.4) is currently implemented implicitly, as loops are simply added first to the trace tree. Default branches (Section 5.2.5) are not yet implemented. In consequence, the `TraceGraph` and `Learner` classes are generic and can be used for any test specification language.

For the input of the test cases and the output of the hypothesis automaton, generic interfaces were defined. In our prototype, both interfaces are implemented using the LTSML format that can be used to represent any type of automaton [Neu09]. For the test cases, we focus on the test specification language *Testing and Test Control Notation (TTCN-3)*, i.e. the `TraceReader` recognizes TTCN-3 keywords and generates traces according to the semantics of TTCN-3.

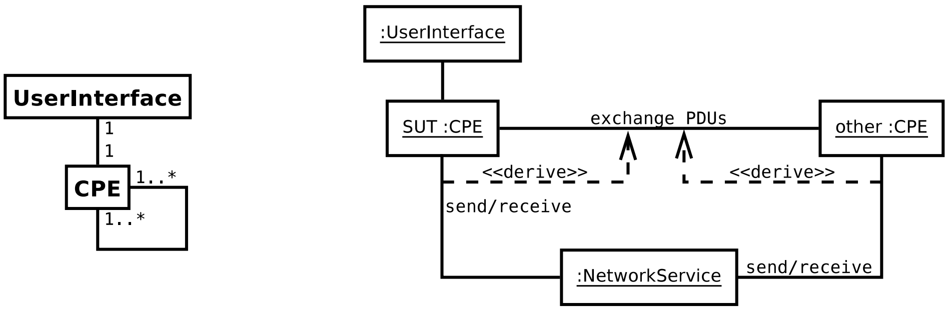
6.2 The Conference Protocol

The conference protocol case study is based on the chat-box program described in [BRS$^+$00]. We have modified it slightly to adapt it to the constraints of our learning procedure.

6.2.1 Description of the Conference Protocol

The conference protocol describes a chat-box program that allows users to participate at a conference chat over a network.

- A user enters an existing conference by sending the service primitive *join*.
- Then, the user can send messages to the conference chat (*datareq*) and receive messages from the conference chat (*dataind*). Each *datareq* causes *dataind* messages to be issued to all other participating users and vice versa.
- At any time after a *join*, the user can leave the conference by sending the service primitive *leave*.



(a) Abstract View of a CPE

(b) Two CPEs Connected over a Network Service

Figure 6.2: Environment of a CPE

The CPEs are responsible to provide this service. They translate the service primitives into *protocol data units (PDUs)* and back. The PDUs are then exchanged through the underlying network.

- A *join* request by the user is forwarded by the local CPE to each of the other participating chat-boxes via *joinPDU*.
- A CPE that receives a *joinPDU* answers
- A *datareq* message causes *dataPDU* to be sent to the other chat-boxes.
- A received *dataPDU* is indicated to the user be *dataind*.
- A *leave* is forwarded as *leavePDU*.

The environment of the CPEs is depicted in Figure 6.2. Every CPE is connected to a user interface, where service primitives are received and sent, and a number of other CPEs, where PDUs are received and sent (Figure 6.2a). The PDUs exchanged between CPEs are transferred via a network service's sent and receive functionality (Figure 6.2b).

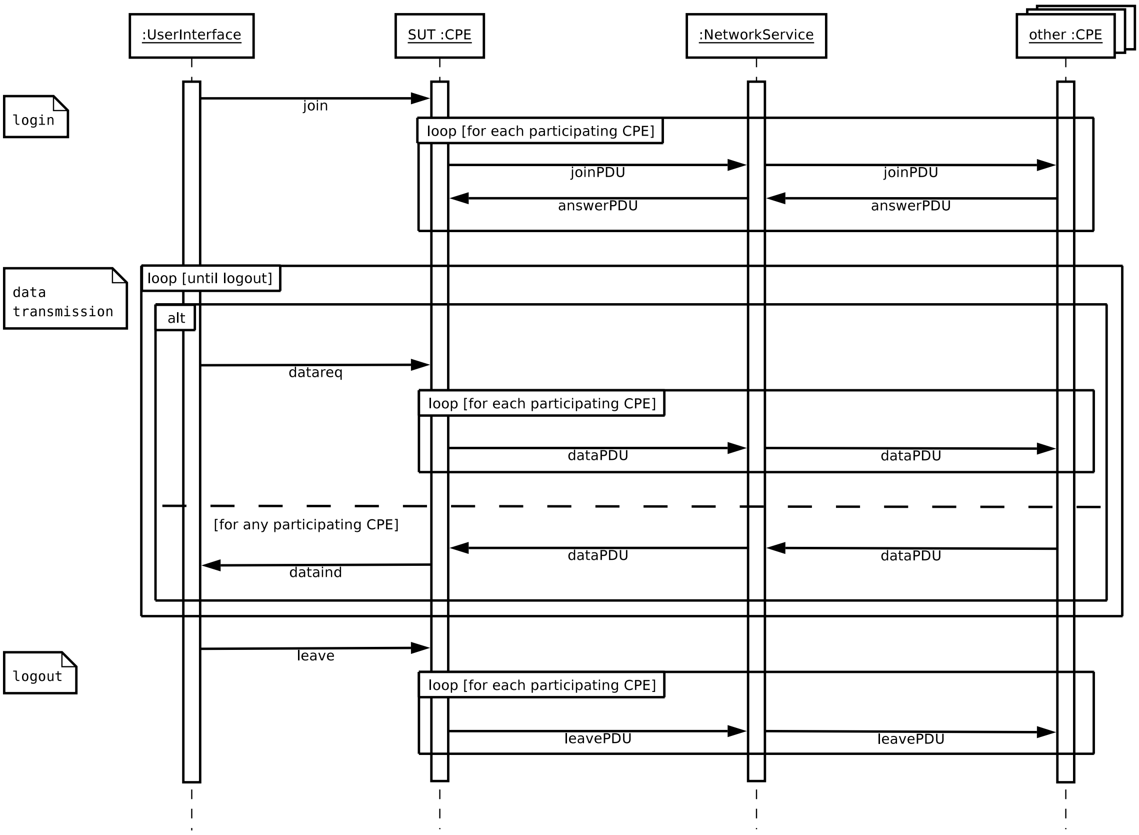


Figure 6.3: Main Flow of the Conference Protocol

The conference protocol can be regarded to consist of three phases (Figure 6.3).

- The *login* phase consists of the user request *join*, which is forwarded to the other participating chat-boxes as *joinPDU* and answered by them via *answerPDU*.
- In the *data transmission* phase, the user can send messages via *datareq*, which are forwarded to all other chat-boxes as *dataPDUout*, and receive messages from all other chat-boxes, which arrive at the CPE as *dataPDUin* and are indicated to the user by *dataind*.
- The last phase is the *logout*, starting with the user's request to *leave*, which is forwarded to the other chat-boxes as *leavePDU*.

6.2.2 Test Scenario for the Conference Protocol

For the test case generation, we assume a test environment as depicted in Figure 6.4, where the SUT is a single CPE and the test environment plays the roles of the user interface ("upper tester") and all other participating CPEs ("lower tester"). The signals sent and received on both interfaces are coordinated by the test driver.

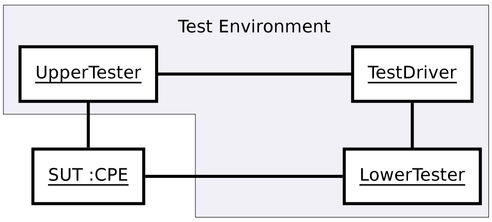


Figure 6.4: Test Environment for a Conference Protocol Entity

In accordance with the need to distinguish between signals sent from the SUT and signals received by the SUT for the learning procedure (Section 4.1.2), we call the outbound *dataPDU* following a *datareq* "*dataPDUout*" and the inbound *dataPDU* causing a *dataind* "*dataPDUin*". In addition, as all PDUs can be received from any of the participating chat-boxes, we append the suffix n to indicate the identification number of the particular chat-box that is the sender or receiver of this PDU, thereby representing the parameterized process via explicit signals.

For all experiments, we assume that the CPE under test enters a conference with a number of other chat-boxes, exchanges an arbitrary number of messages, and leaves again. In this scenario, no other CPEs can enter or leave the chat during the observation period. All test cases for the protocol are therefore designed to ensure the following:

- A CPE cannot enter the data transmission phase before all participating chatboxes have answered the *join* request, i.e. all *joinPDU* have been issued and all *answerPDU* have been received.
- Data transmission is regarded as uninterruptible, therefore all *dataPDUout* have to be issued before another *datareq* can be processed or a *dataPDUin* can be received.

6.3 Learning the Conference Protocol: Fixed Signal Sequence

For the first experiment, we consider a restricted version of the conference protocol. We assume a reliable medium service where no messages are lost, and where the sequence of messages is preserved. Also, we presume that the CPE under test sends all *joinPDU* before receiving the *answerPDU* from the other participating CPEs. In consequence, we expect the automaton representing the conference protocol to resemble the one in Figure 6.5. In Figure 6.5, we have abstracted from the number of participating CPEs in the following way. The dotted edges represent a sequence of PDU, depending on the number of CPE participating in the chat. For the dashed edges, one transition exists for every CPE participating in the chat.

As an example, the target automaton for two participating CPEs is given in Figure 6.6. In this graph, the generic edges from Figure 6.5 are replaced by the according concrete edges. For every dotted edge, there is a sequence of edges, e.g. the dotted edge “send dataPDU to all other CPEs” is replaced by the sequence dataPDUout_1, dataPDUout_2. The dashed edge is replaced by two edges: instead of the generic edge “receive dataPDU from any other CPEs” there are the two edges dataPDUin_1 and dataPDUin_2.

6.3.1 Test Suite 1: No Specified Cycles

We specify the first test suite for the conference protocol without explicitly declaring cycles. In consequence, the trace graph also does not contain cycles. Instead, we build the test cases to satisfy a boundary-interior coverage, where the cycles in the data transmission phase of the protocol are executed once or twice, or skipped.

Table 6.1 shows the results of the experiment. We scale the experiment according to the number of CPEs participating in the chat besides the CPE under test. This number is shown in the first column of the table. In addition, we compare the number of *pass* traces described in the test suite, the size of the trace graph, and the size of the target

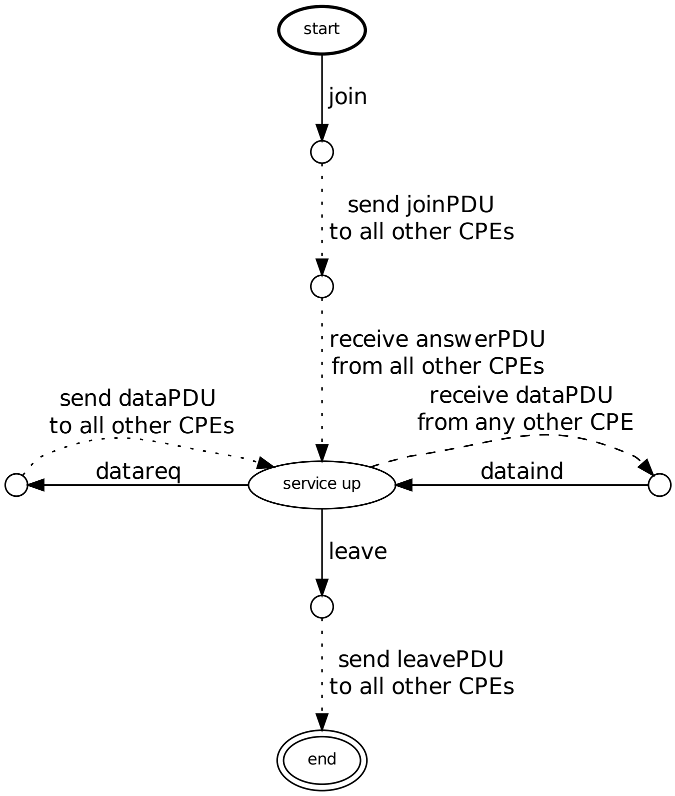


Figure 6.5: Generic Target Automaton for the Simplified Conference Protocol

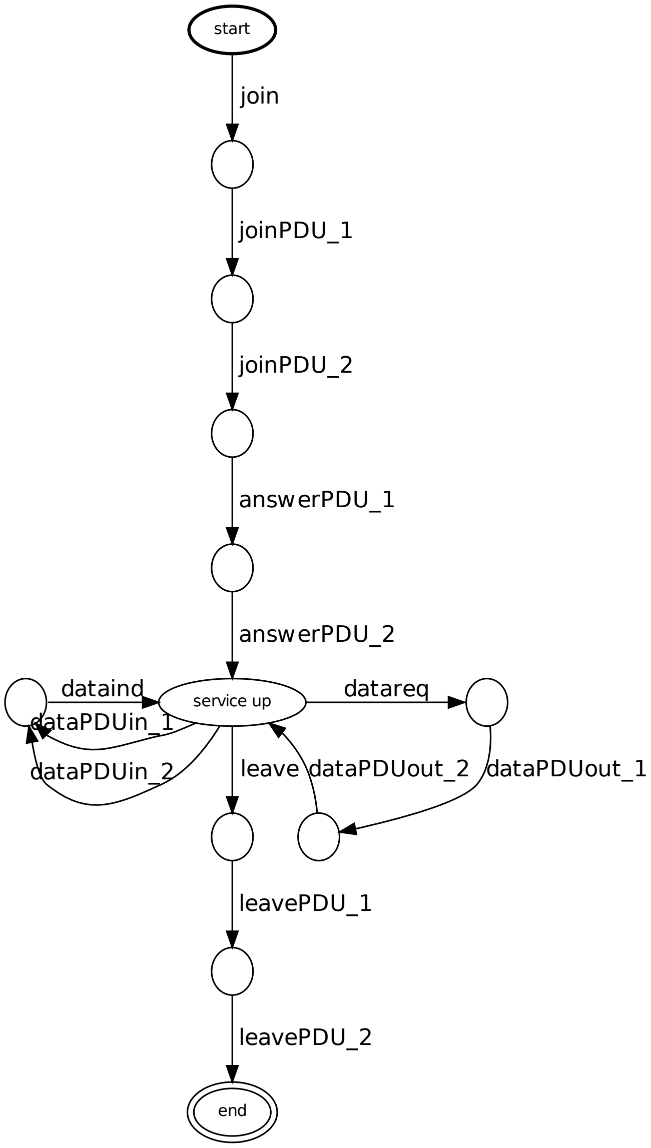


Figure 6.6: Target Automaton for the Simplified Conference Protocol: Two Participating CPEs

automaton. For every number of CPEs, the learned automaton matches the expected automaton shown in Figure 6.5. Representatively, we show the learned automaton for the scenario with two participating CPEs in Figure 6.7. The time listed in the last column of Table 6.1 was measured by an *integrated development environment (IDE)* and therefore only represents an estimation.

CPEs	Pass Traces	Trace Graph	Learned Automaton	Time
1	6 (3)	33 (20) nodes	72 edges, 8 nodes	1 second
2	9 (4)	60 (37) nodes	168 edges, 12 nodes	2 seconds
3	12 (5)	90 (52) nodes	304 edges, 16 nodes	3 seconds
4	15 (6)	120 (75) nodes	480 edges, 20 nodes	5 seconds
5	18 (7)	164 (97) nodes	696 edges, 24 nodes	12 seconds

Table 6.1: Results for Test Suite 1

The inspection of the learner's logfile shows that the longest traces queried only expand any cycle once. Therefore, the simple conference protocol can also be learned from a smaller test suite, which contains traces representing the cycles in the data transmission phase plus one trace where the CPE under test leaves immediately after login. As the smaller test suite contains shorter traces, the trace graph is also smaller. The size of the trace graph and the number of test cases for the reduced test suite are shown in brackets in Table 6.1. However, we believe that this result is due to the structure of the protocol, where all cycles start in the same state, and cannot be generalized to other systems.

6.3.2 Test Suite 2: Specified Cycles

In the second test suite, we explicitly declare cycles so that they can be represented in the trace graph. As before, we scale the experiment according to the number of CPEs participating in the chat besides the CPE under test and every learned automaton matched the expected automaton shown as in Figure 6.5. In fact, the learned automata were exactly the same as the automata learned in Section 6.3.1, therefore we omit showing an according automaton here. The numerical results are shown in Table 6.2.

Comparing Tables 6.1 and 6.2, we observe that the time needed for learning is almost the same for both test suites. Also, both test suites produce the same automata. The main difference is that in the first experiment, a larger test suite is needed. In consequence, the trace graph representing the test suite is also larger. Also, both the size of the test

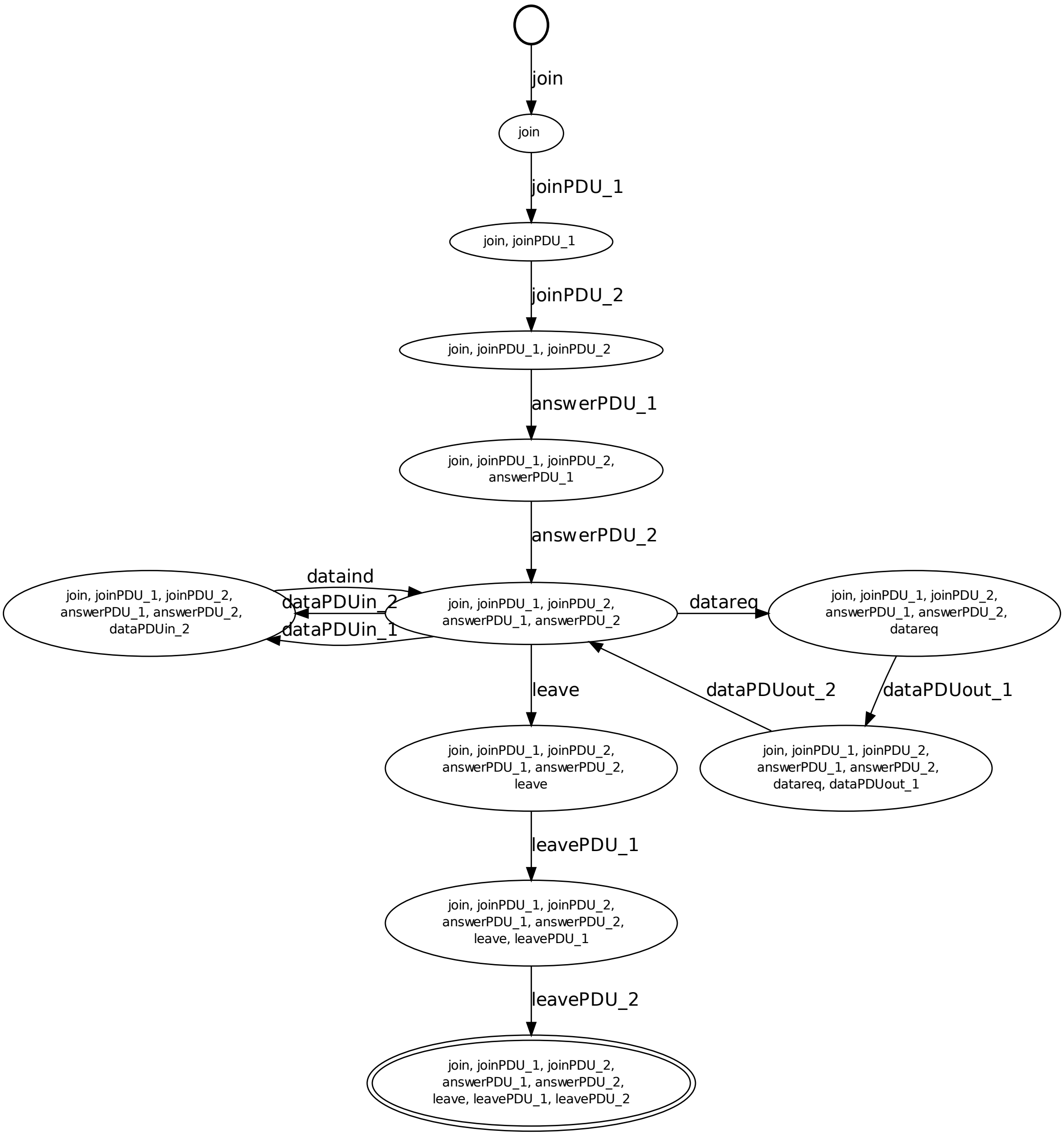

join
join
joinPDU_1
join, joinPDU_1
joinPDU_2
join, joinPDU_1, joinPDU_2
answerPDU_1
join, joinPDU_1, joinPDU_2, answerPDU_1
answerPDU_2
join, joinPDU_1, joinPDU_2, answerPDU_1, answerPDU_2
dataind
dataPDUin_2
dataPDUin_1
join, joinPDU_1, joinPDU_2, answerPDU_1, answerPDU_2, dataPDUin_2
datareq
join, joinPDU_1, joinPDU_2, answerPDU_1, answerPDU_2, datareq
dataPDUout_1
join, joinPDU_1, joinPDU_2, answerPDU_1, answerPDU_2, datareq, dataPDUout_1
dataPDUout_2
leave
join, joinPDU_1, joinPDU_2, answerPDU_1, answerPDU_2, leave
leavePDU_1
join, joinPDU_1, joinPDU_2, answerPDU_1, answerPDU_2, leave, leavePDU_1
leavePDU_2
join, joinPDU_1, joinPDU_2, answerPDU_1, answerPDU_2, leave, leavePDU_1, leavePDU_2

Figure 6.7: Automaton Learned from Acyclic Test Cases

CPEs	Pass Traces	Trace Graph	Learned Automaton	Time
1	2	13 nodes	72 edges, 8 nodes	1 second
2	3	22 nodes	168 edges, 12 nodes	2 seconds
3	4	30 nodes	304 edges, 16 nodes	3 seconds
4	5	40 nodes	480 edges, 20 nodes	5 seconds
5	6	48 nodes	696 edges, 24 nodes	13 seconds

Table 6.2: Results for Test Suite 2

suite and the size of the trace graph increase faster in proportion to the number of CPEs than for the second test suite.

6.4 Learning the Conference Protocol: Variable Signal Sequence

In the following, we examine a more complicated version of the conference protocol. While we also assume that no signals are lost by the medium service, the sequence the signals are sent in is not preserved, so that the PDUs may be observed in any sequence. To limit the complexity, we assume that all *joinPDU* are sent before the first *answerPDU* is received. Based on these assumptions, we expect a model like the one shown in Figure 6.8. As in Section 6.3, the dashed edge stands for a cluster of edges, containing one edge for every participating CPE. Each of the dotted edges represents a cluster of signal sequences, containing an edge for every possible sequence of the respective PDUs. As the number of PDUs sent along each of the dotted edges depends on the number of participating CPEs, so does the number of different serializations. In fact, if |CPE| denotes the number of participating CPEs, then there are |CPE| different PDUs of each type, one to or from every other CPEs, and (|CPE|)! permutations of every series of PDUs.

An example is given in Figure 6.9. For every dotted edge in Figure 6.8, there are two sequences of edges in Figure 6.9: e.g. the dotted edge “send dataPDU to all other CPEs” is replaced by the sequences dataPDUout_1, dataPDUout_2 and dataPDUout_2, dataPDUout_1. As in Section 6.3, the dashed edge is replaced by two edges: instead of the generic edge “receive dataPDU from any other CPEs” there are the two edges dataPDUin_1 and dataPDUin_2.

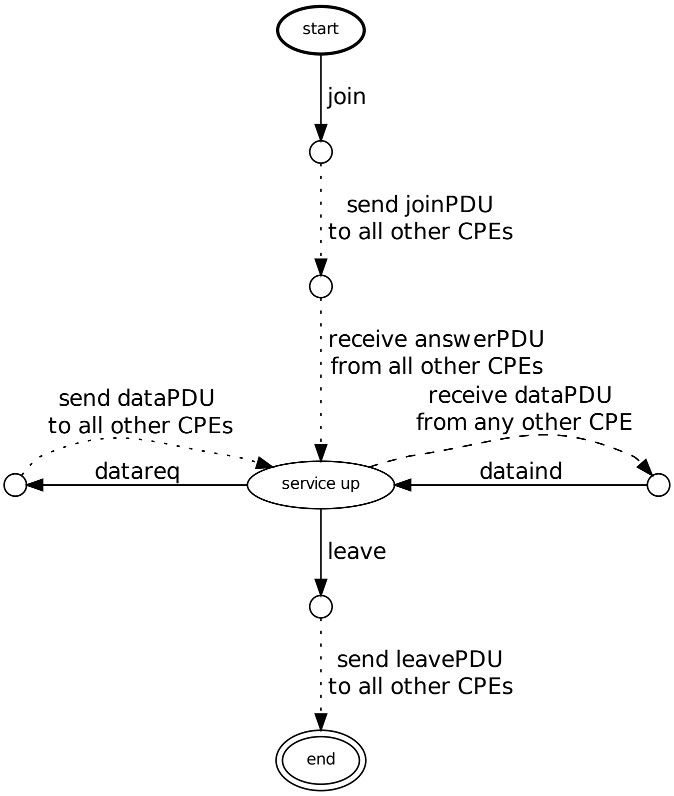


Figure 6.8: Generic Target Automaton for the Conference Protocol

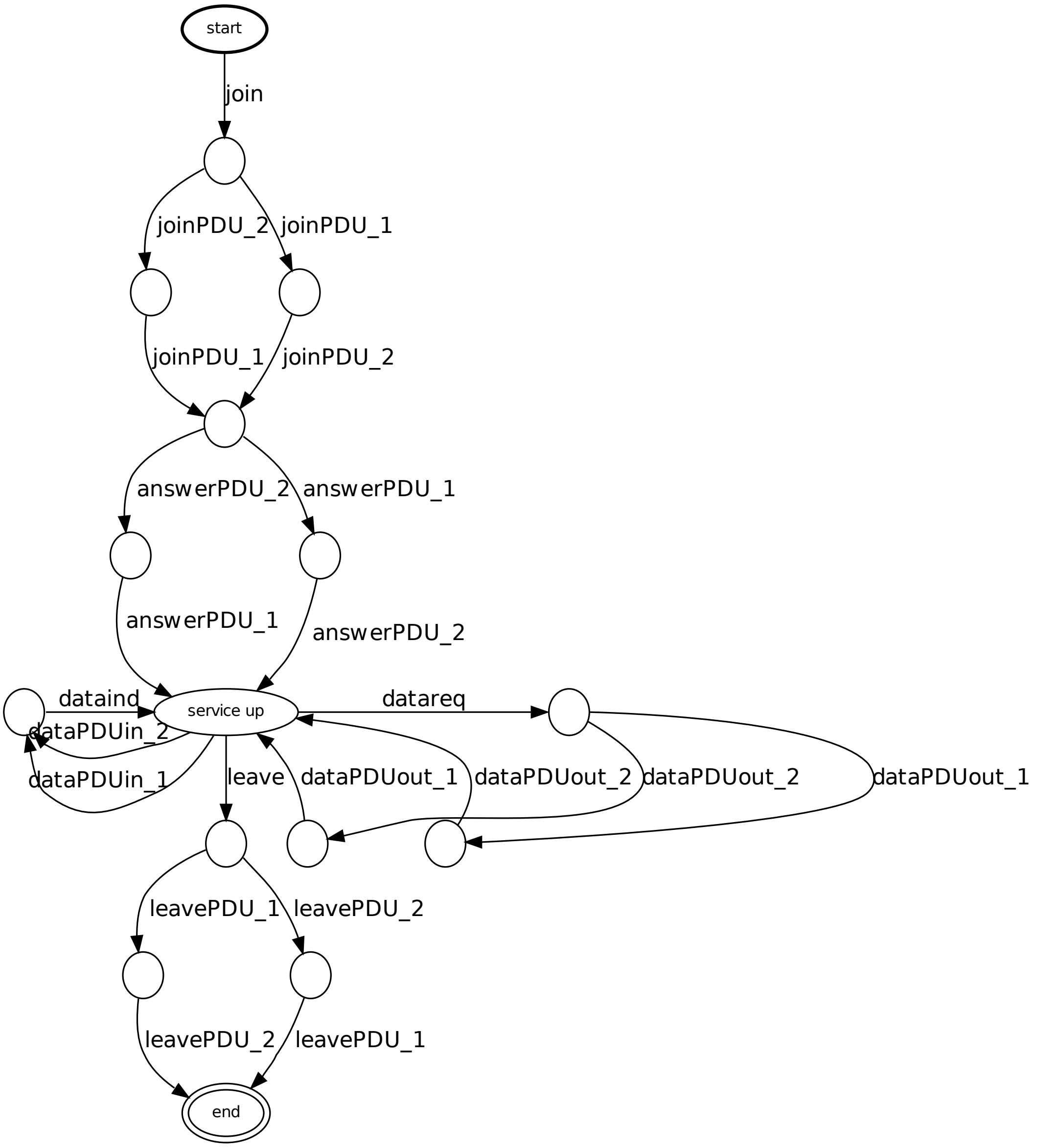


Figure 6.9: Target Automaton for the Conference Protocol: Two Participating CPEs

6.4.1 Test Suite 3: Branch Coverage

A common coverage criterion in testing is the branch coverage, where every branch of the SUT is executed. As the problem of messages arriving out of order affects mostly the PDUs, this means that we have to generate a trace for every permutation of the four series of PDUs that are sent in the protocol, *joinPDU*, *answerPDU*, *dataPDUout*, and *leavePDU*, as the according service primitives *join*, *datareq*, and *leave* each cause a series of PDUs to be sent. For the sequences of PDUs, the branch coverage implies that while every possible serialization of PDUs is represented at least once, the serializations are recombined among themselves. The cycles in the data transmission phase of the protocol are specified explicitly.

CPEs	Pass Traces	Trace Graph	Learned Automaton	Time
1	2	13 nodes	72 edges, 8 nodes	0 seconds
2	4	37 nodes	294 edges, 21 nodes	1 seconds
3	9	134 nodes	1368 edges, 72 nodes	5 seconds
4	28	659 nodes	7848 edges, 327 nodes	10 minutes

Table 6.3: Results for Test Suite 3

While the numerical results shown in Table 6.3 look quite encouraging, as they show that rather large automata can be learned in acceptable time. However, a closer look at the learned automata quickly discovers that the learned automata do not match the target automaton as shown in Figure 6.8. Instead of a compact automaton, the learned automaton contains the traces exactly as they were specified in the test cases. As an example, the automaton learned for a chat session with two CPEs besides the CPE under test is shown in Figure 6.10. Instead of generalizing from the input data, the learning algorithm learned every input trace by heart.

6.4.2 Test Suite 4: Path Coverage

Since the test suite described in Section 6.4.1 did not reproduce the expected automaton, we generate another test suite for the complex variation of the conference protocol, this time satisfying path coverage of the expected automaton. Therefore, in addition to representing all permutations of every series of PDUs, we also include every combination of those permutations in the test suite. Again, the cycles in the data transmission phase of the protocol are specified explicitly.

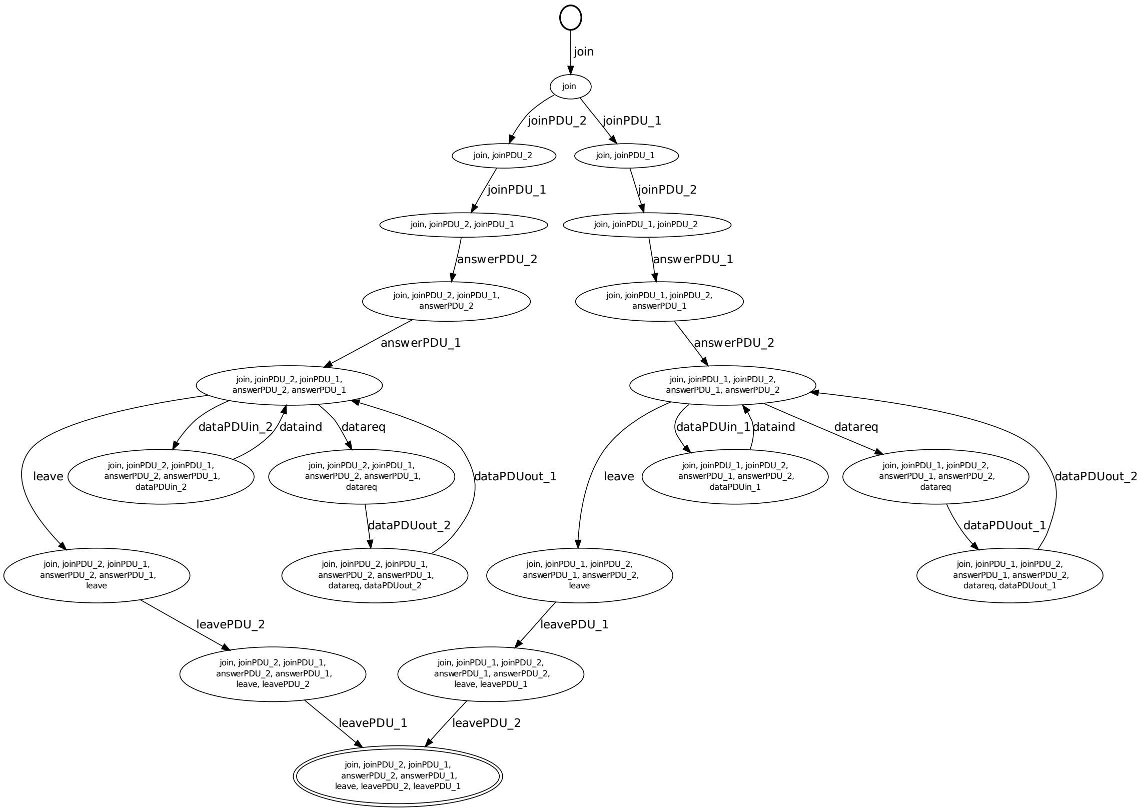

join
join
joinPDU_2
joinPDU_1
join, joinPDU_2
join, joinPDU_1
joinPDU_1
joinPDU_2
join, joinPDU_2, joinPDU_1
join, joinPDU_1, joinPDU_2
answerPDU_2
answerPDU_1
join, joinPDU_2, joinPDU_1, answerPDU_2
join, joinPDU_1, joinPDU_2, answerPDU_1
answerPDU_1
answerPDU_2
join, joinPDU_2, joinPDU_1, answerPDU_2, answerPDU_1
join, joinPDU_1, joinPDU_2, answerPDU_1, answerPDU_2
dataPDUin_2
dataind
datareq
dataPDUin_1
dataind
datareq
leave
join, joinPDU_2, joinPDU_1, answerPDU_2, answerPDU_1, dataPDUin_2
join, joinPDU_2, joinPDU_1, answerPDU_2, answerPDU_1, datareq
dataPDUout_1
leave
join, joinPDU_1, joinPDU_2, answerPDU_1, answerPDU_2, dataPDUin_1
join, joinPDU_1, joinPDU_2, answerPDU_1, answerPDU_2, datareq
dataPDUout_2
dataPDUout_2
dataPDUout_1
join, joinPDU_2, joinPDU_1, answerPDU_2, answerPDU_1, leave
join, joinPDU_2, joinPDU_1, answerPDU_2, answerPDU_1, datareq, dataPDUout_2
join, joinPDU_1, joinPDU_2, answerPDU_1, answerPDU_2, leave
join, joinPDU_1, joinPDU_2, answerPDU_1, answerPDU_2, datareq, dataPDUout_1
leavePDU_2
leavePDU_1
join, joinPDU_2, joinPDU_1, answerPDU_2, answerPDU_1, leave, leavePDU_2
join, joinPDU_1, joinPDU_2, answerPDU_1, answerPDU_2, leave, leavePDU_1
leavePDU_1
leavePDU_2
join, joinPDU_2, joinPDU_1, answerPDU_2, answerPDU_1, leave, leavePDU_2, leavePDU_1

Figure 6.10: Resulting Automaton for Transition Coverage

Now, the expected automaton can be learned correctly from the test suite, as demonstrates the resulting automaton for the scenario of two other CPEs shown in Figure 6.11. Unfortunately, a test suite satisfying path coverage grows very fast with increasing number of participating CPEs, as the number of permutations is factorial in the number of CPEs, |CPE|!, and the number of combinations can be estimated as a multiplication of all permutations, $|\text{CPE}|!^4$. So, while the test suite for two participating CPEs comprises a manageable number of 32 *pass* traces, the test suite for three participating CPEs already includes 1944 *pass* traces, which surpasses Java's heap capacity on the experimentation machine.

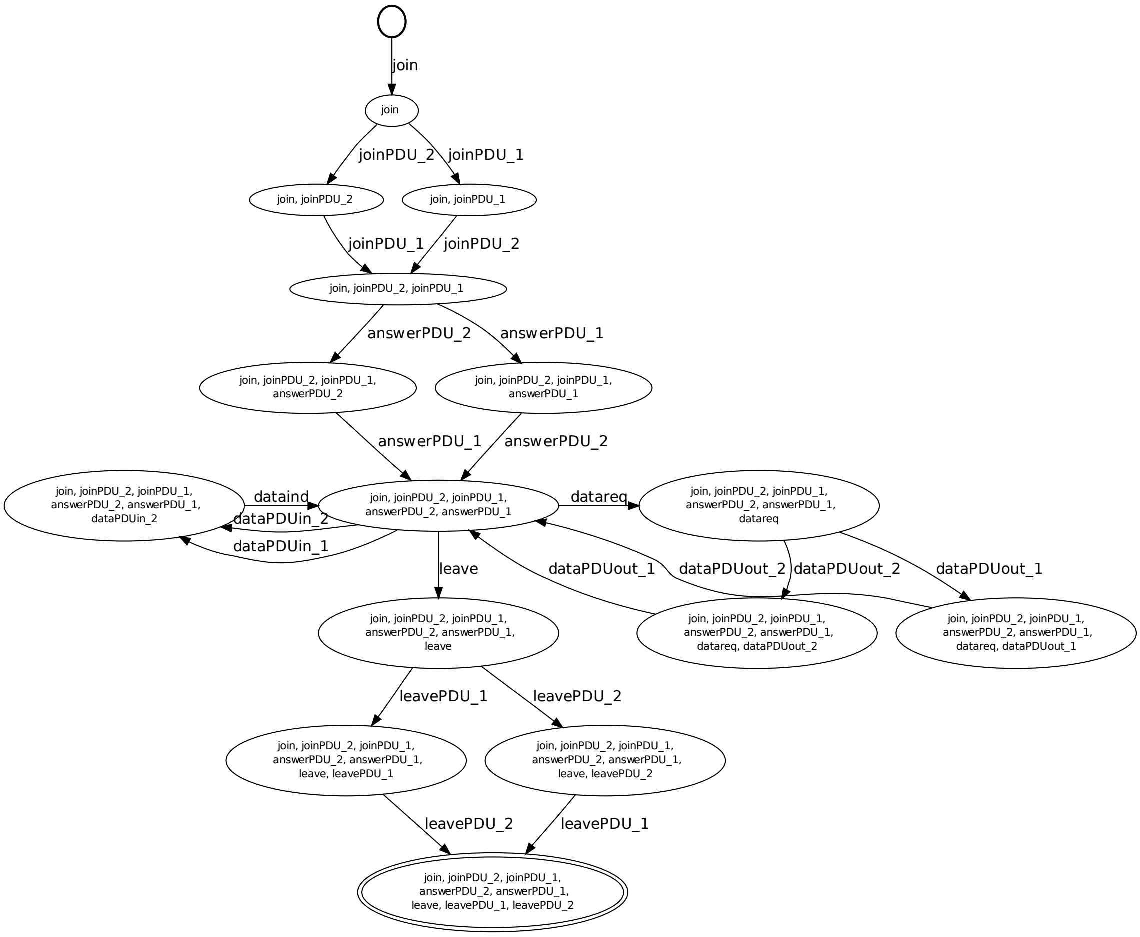

join
join
joinPDU_2
joinPDU_1
join, joinPDU_2
join, joinPDU_1
joinPDU_1
joinPDU_2
join, joinPDU_2, joinPDU_1
answerPDU_2
answerPDU_1
join, joinPDU_2, joinPDU_1, answerPDU_2
join, joinPDU_2, joinPDU_1, answerPDU_1
answerPDU_1
answerPDU_2
join, joinPDU_2, joinPDU_1, answerPDU_2, answerPDU_1, dataPDUin_2
dataind
dataPDUin_2
dataPDUin_1
join, joinPDU_2, joinPDU_1, answerPDU_2, answerPDU_1
datareq
join, joinPDU_2, joinPDU_1, answerPDU_2, answerPDU_1, datareq
leave
dataPDUout_1
dataPDUout_2
dataPDUout_2
dataPDUout_1
join, joinPDU_2, joinPDU_1, answerPDU_2, answerPDU_1, leave
join, joinPDU_2, joinPDU_1, answerPDU_2, answerPDU_1, datareq, dataPDUout_2
join, joinPDU_2, joinPDU_1, answerPDU_2, answerPDU_1, datareq, dataPDUout_1
leavePDU_1
leavePDU_2
join, joinPDU_2, joinPDU_1, answerPDU_2, answerPDU_1, leave, leavePDU_1
join, joinPDU_2, joinPDU_1, answerPDU_2, answerPDU_1, leave, leavePDU_2
leavePDU_2
leavePDU_1
join, joinPDU_2, joinPDU_1, answerPDU_2, answerPDU_1, leave, leavePDU_1, leavePDU_2

Figure 6.11: Resulting Automaton for Path Coverage

CPEs	Pass Traces	Trace Graph	Learned Automaton	Time
1	2	13 nodes	72 edges, 8 nodes	0 seconds
2	32	90 nodes	224 edges, 16 nodes	1 seconds

Table 6.4: Results for Test Suite 4

6.5 Conclusion of the Case Study

Our experiments have shown that it is possible to reconstruct an automaton model from a test suite. We have performed four experiments, generating two different test suites for each of two versions of the conference protocol. In three experiments, the target automaton was correctly identified. However, in all successful experiments, the test suite satisfied a path coverage on the target automaton. The observed computing times and automata sizes suggest that while the learner is able to handle quite large automata, the size of the required test suite is the inhibiting factor.

In the first version of the conference protocol, especially the signals sequences used in the setup and shutdown phase were fixed. In consequence, the learner quickly recognized the main loop of the conference protocol, even when cycles were not explicitly marked in the test cases. Actually, while both test suites for the simple version of the conference protocol were designed to satisfy branch coverage of the expected automaton, due to the simple structure of the expected automaton, both test suites also satisfy a path coverage limited in the number of cycle executions.

In the second version of the conference protocol, we introduced a wide range of alternative paths through the protocol automaton. While the resulting automaton should have been quite compact, allowing for the recombination of different partial paths through connecting states, the learner only recognized this structure when examples for all possible paths were given, leading to a rapid growth of the test suite in relation to the increasing number of participating *conference protocol entities (CPEs).*

Comparing the two versions of the conference protocol, we deduce that the structure of the *system under test (SUT)* has an influence on the complexity of the learning process and that for correct machine learning, the test suite has to be as complete as possible. In fact, this precondition on the learning sample has been described before as the need for a "structurally complete" sample. As a rule of thumb, we might say that "the larger the test suite, the smaller the automaton", as a large test suite usually allows more possible paths.

Experimentation has shown that the growing size of the test suite affects our learning procedure in two related points. The first is the generation of the trace graph. For larger test cases the preprocessing steps, such as cycle detection, are harder to handle. The second is the equivalence query, where the whole test suite has to be executed again and again. Interestingly, it is not necessary to compute all interleaving cycle executions for the equivalence query. Instead, the crucial points for the equivalence query turn out to be the intersections between different paths through the SUT. Therefore, for a better scalability of our learning procedure, we should aim at detecting intersections of the test cases in the trace graph, thereby minimizing the trace graph and reducing the necessary size of the test suite while keeping its expressiveness.

7 Discussion and Open Questions

While the case study, described in Chapter 6, proved the suitability of the learning approach, it also raised a number of questions regarding specific properties of the used data structures and the algorithm. Some of the observations confirmed design decisions, while at other points, decisions turned out to be less then optimal. The following sections provide an assessment of the parts of our learning approach. We reassess the generated automaton, the use of a test suite as input sample, and the learning algorithm itself with respect to their suitability for our purposes. As some of the encountered questions have also attracted the attention of other researchers, there are a number of possible solutions available that could be adapted to our learning approach. In other situations, a number of possible solutions are suggested that have not been investigated yet.

7.1 Suitability of the Learned Automaton

Angluin's learning algorithm generates a plain *deterministic finite automaton (DFA)*, which consists of a set of states partitioned into accepting and rejecting states, an input alphabet triggering state transitions and a corresponding state transition relation. As we have argued in Section 4.1.1, this type of automaton is suitable for representing system models. The drawback of DFAs is that to express the same information as a more advanced model, more states are needed, making the automaton large. Contrary to expectations, it was not the size of the target DFA that proved to be a problem, but its structure and the repercussions on the size of the sample needed to correctly reconstruct the automaton.

7.1.1 Influence of Parameters

A DFA representing a parameterized process such as the conference protocol in Section 6.4 contains a number of paths to cover different serializations of the parameters. The experiments show that to correctly learn all the different serializations, not only all of them need to be represented in the learning sample, but also in every possible combination. This also correlates to the results by Berg et al. regarding prefix-closed models [BJLS05], which state that prefix-closed automata are harder to learn than random automata, as the learning algorithm would need to perform a membership query for every prefix.

Berg et al. address this problem by proposing an approach to learn parameterized automata [BJR06]. Based on the original version of Angluin's learning algorithm, which uses an observation table to store the discovered information, they introduce a guard-labeling on the entries of the table, describing an input partitioning. Then, the result of an equivalence query can also be the splitting of a partition beside the discovery of a new state or the acceptance of the learned automaton. A similar approach is described by Li et al. [LGS06], which has the advantage of taking into account both input and output signals, where Berg et al. only consider input signals.

Both proposed solutions for learning parameterized models rely on the original version of Angluin's learning algorithm, which uses an observation table to store the information learned, and the table format is essential in computing the parameterization. In contrast, our approach to learning from test cases uses a variation introduced by Kearns and Vazirani [KV94], which stores the gathered information in a classification tree. Therefore, an adoption of those solutions requires some adaptations.

7.1.2 Handling Non-Applicable Signals

Another structural problem of the learned DFA is that the learning algorithm always generates a fully specified automaton, i.e. an automaton where in every state, a target state for every possible signal is specified. As the essence of state based systems is that the applicable actions depend on the state of the system, most states of an automaton can only process a subset of the global signal alphabet. The handling of the non-applicable signals then depends on the semantics of the system. One commonly adopted approach is that the system should robustly ignore all unspecified signals, which implies that unspecified transitions at a given state are treated as self-loops.

However, this approach cannot be adopted by Angluin's learning algorithm, as the algorithm only discerns accepted and rejected traces and therefore cannot tell whether a signal is not specified and should be ignored via a self-loop or whether a signal is explicitly rejected and should lead to a fail state. In consequence, the learning algorithm routes all unspecified or rejected signals into a global fail state (Section 4.1.1), thereby generating a fully specified automaton that rejects non-applicable signals.

Due to the properties of the learning algorithm, we can identify the global fail state in the learned automaton, as it is the first rejecting state discovered. Therefore, it would be possible to remove the global fail state and to replace transitions leading into it by self-loops to their source states. This is, however, not a safe transformation, as thereby all explicitly failing transitions would also be transformed into self-loops. In consequence, to learn a DFA which ignores some inopportune signals, those self-loop transitions need to be explicitly specified in the test suite. This obviously leads to a larger test suite, which is also less intuitive and less readable. Alternative approaches would be to distinguish explicitly and implicitly rejected transitions during learning, generating self-loops for implicitly rejected transitions, or to implement a smart transformation algorithm which checks transitions to the global fail state before removing them.

7.2 Suitability of a Test Suite as Sample Data

The main idea of our learning approach was to use a test suite as input data, as the verdicts *pass* and *fail* readily provided an assessment of acceptable and rejectable system traces. While this assumption holds true, mapping test cases to input traces of Angluin's learning algorithm nevertheless reduces the expressiveness of the test cases considerable. As described in Section 4.1.2, the test cases need to be linearized, a common starting state has to be established, and all circumstantial information as parameters and ports have to be integrated into the input of the target automaton. Especially the flattening of parameters and ports leads to an exponential blowup of the number of test case traces.

While Angluin's algorithm depends on traces, the semantic state-merging approach introduced in Chapter 5 is designed to exploit the test language specific properties of test cases. The test language specific back-end then represents the sample data in a generic way to the learning algorithm. This way, the learning algorithm provides a common front-end to be combined with different test language specific back-ends. In consequence, further optimization regarding the representation of the test suite mainly concerns the state-merging part of our hybrid algorithm.

7.2.1 Mining Additional Properties

The state-merging techniques introduced in Section 5.2 define how to merge traces by generating a prefix tree, the representation and handling of cycles in the trace graph and the handling of default branches. However, cycles and defaults are only the most common properties of test languages.

Stable testing states define known and checkable states of the *system under test (SUT)*. By marking the according states in the trace graph, a test case containing a marked testing state could be directly connected at the given state. Thereby, the need for a common starting state could be avoided.

Parallel behavior can be explicitly defined, especially in test languages that are targeted at distributed testing. By defining an according operator on the trace tree, membership queries containing different sequentializations of parallel behavior could be answered correctly without explicitly representing every such path in the test suite.

Besides the verdicts *pass* and *fail*, some test languages define additional verdicts assigned on inconclusive behavior or on an error in the test environment. In the current learning approach, everything not accepted, i.e. assigned a *pass* verdict, is rejected. However, in an open world approach, the answer “I don’t know” could be given by the membership oracle. In this case, the mapping of the test verdicts has to be reconsidered. The verdict *inconc*, which is used by *Testing and Test Control Notation (TTCN-3)* to indicate that the result of the test case cannot be decided, maps naturally on an “I don’t know” for the learner—the teacher doesn’t know whether the behavior is acceptable or not. Then again, the verdict *error* is considered by TTCN-3 to be more severe than a verdict *fail*, but for learning purposes it could still amount to an “I don’t know”.

Lastly, information about *ports* in the test cases could also be used. Considering a highly connectable SUT, such a system would feature a number of different ports connecting to different other systems. To learn a protocol automaton for just a subset of those communication ports, the test case traces could be restricted to the ports in question, excluding all others.

7.2.2 Influence of Coverage

While the semantic state-merging approach is able to make up for many missing traces, the case study suggests that the test suite used in learning must at least satisfy a path coverage of the SUT, as the one experiment where only a branch coverage was used failed. However, there are other coverage criteria besides branch and path coverage, e.g. based on condition determination or on the functions of the SUT, or automatic test case generation techniques. Further research is needed to clarify the dependencies of system structure, test suite coverage, and learnability.

7.3 Suitability of the Learning Algorithm

When confronted with the problem of reconstructing a system model from test cases, learning algorithms seemed to be a simple solution. The starting assumption was that while the test cases could be used as they were, some adaptations would have to be made to the algorithm. Instead, research shows that the learning algorithm itself can be used without changes, while the effort of adaptation concerns the representation of the learning sample, i.e. the test cases. In fact, the approach to learning from test cases proved to be a problem of teaching more than of learning.

7.3.1 Online and Offline Learning

Online learning algorithms, like Angluin's algorithm, build a system model by querying a teacher. Their main advantage is in generating the necessary queries themselves, thereby avoiding the need for a complete sample. However, this approach implies the existence of an omniscient oracle, which is able to answer arbitrary queries. In contrast, offline learning algorithms assume the existence of a structurally complete sample, which is then merged into the target automaton.

The query mechanisms used by Angluin's algorithm naturally match the test cases' verdicts. Also, Angluin's algorithm is scalable, growing only linearly with the size of the target automaton, and always generates a minimal DFA. As a test suite always assumes completeness with regard to certain coverage criteria, it can be assumed that the completeness of the test suite is sufficient to answer the membership queries. However, our experiments show that this assumption holds only for high coverage criteria and even then depends on the structure of the system (Section 7.1).

These results seem to suggest that an offline learning approach would work better for the learning from test cases. Lambeau et al. [LDD08] propose an offline algorithm based on state-merging, which takes into account merge constraints as well as incompatibility constraints, requiring some states to be merged obligatorily while others need to stay separated. This approach might work well for the learning from test cases. However, state-merging algorithms always need to be closely tailored to the sample data to merge. Therefore, a state-merging approach would only work for the special semantics it is designed for.

Our approach combines the advantages of both online and offline algorithms. The online algorithm is used to drive the overall learning process, thereby establishing a learning procedure which is independent from any given test specification language.

Underlying the learning process, state-merging is used to mine the test language specific information for better coverage of the automaton's traces and to generate a data structure to be used as an oracle.

Following this layered approach, existing methods could be integrated for optimization. Regarding the online learning part, these optimizations concern the type of automaton generated, incorporating i.e. parameterization [BJR06, LGS06]. Optimizations on the offline learning part should extend the semantic state-merging approach introduced in Chapter 5. Possibly exploitable properties of test cases comprise differentiation of input and output signals and consideration of stable testing states. For example, the stable testing states could be matched to the merge constraints in the approach by Lambeau et al. [LDD08].

7.3.2 Other Versions of Angluin's Algorithm

Another source for optimization of the learning procedure is the version of Angluin's algorithm that is used. The prototypical implementation uses a variation introduced by Kearns and Vazirani [KV94], which stores the gathered information in a classification tree (Section 3.7). This version has the advantage of asking less membership queries, but at the cost of more equivalence queries [BDGW97]. Also, the classification tree provides a structure which is easy to maintain and therefore quickly to implement.

The original version of Angluin's algorithm uses an observation table to store the gathered information. This variation asks more membership queries before constructing the first hypothesis automaton, thereby reducing the number of needed equivalence queries [BDGW97]. On the other hand, maintaining the consistency of the observation table needs more effort.

Experimentation shows that using the trace graph as an oracle, membership queries are cheap as their complexity only depends on the length of the queried trace, while equivalence queries take time as in the worst case, the whole test suite has to be run against the hypothesis automaton (Section 5.3.2). Also, the adaptations to learning from test cases are completely independent of the underlying implementation of Angluin's algorithm. Therefore, re-implementing the core of the learning algorithm according to the original version of Angluin's algorithm might even provide a small performance advantage.

7.3.3 Breaking the Closed World Assumption

In most learning scenarios, it is comparatively easy to get a correct answer to membership queries, while the equivalence query is hard to decide. When learning from a complete test suite, it is the other way around. The equivalence query can be matched easily to a run of the test suite against the hypothesis automaton (Definition 29), the only limiting factor being the time needed to run a large test suite. This also relates to the results of Berg at al. [BGJ$^+$05], who investigate the similarities between conformance testing and automata inference, finding that conformance testing solves a checking problem. Therefore, we can safely assume that a test suite that is sufficient to declare a system as conforming to its specification also suffices to decide whether it is equivalent to a learned hypothesis automaton.

In contrast, when asking membership queries against a limited set of traces, as every test suite is bound to be, there will always be queried traces that are not contained in the test suite. As the experiments show, rejecting every unknown trace can lead to bad generalization in the hypothesis automaton (Section 6.4.1), while trying to provide for every possibly query leads to inhibitively large test suites (Section 6.4.2). There are several possible approaches to solve this dilemma.

One way is to mine the test suite for implicit traces by state-merging. First efforts in this direction have been integrated into our learning approach (Chapter 5) and have been proven effective by the case study. Besides further exploitation of the properties of the test languages, also input from existing research in state-merging techniques can be used. The state-merging approach has two main advantages. The approach is self-contained, as no external input is needed, and it is safe, as the state-merging is based on information internal to the test suite. The drawback is that the mining depends on the information available in the test suite. If a test language with restricted description possibilities is used, the possibilities of state-merging are also restricted. Besides, while state-merging is able to boost the number of covered traces, it cannot make up for missing test cases if the test suites coverage is insufficient.

Another possible approach is to include explicit "don't know" into the possible answers of a membership query. The problem of undecidable membership queries has occurred to researchers in other settings before, therefore a number of possibly adaptable methods exist. Sloan and Turán [ST94] define a meta-algorithm for incomplete membership oracles. For each undecidable membership query, the learning process is forked, one instance assuming acceptance, the other rejection of the queried string. If a copy is detected to be inconsistent, it is pruned. While this approach clearly leads to an exponential growth in the computation, the difficulty also is how to determine inconsistencies in the forked hypotheses. Bshouty and Owshanko [BO01] propose an extension of Angluin's learning algorithm including "don't know" as a possible answer to a membership query. Based on the Kearns-Vazirani version of the algorithm, they partition the possible traces of the target automaton into cover sets, resetting the algorithm when a counter-example causes a cover set to be split. Grinchtein and Leucker [GL06] also suggest an extension of Angluin's algorithm. Using the Angluin's original version, they generate an incomplete observation table which they subsequently feed into a satisfiability solver, filling in the gaps with the most consistent solution. However, all those approaches share the disadvantage of replacing the uncertainties of the membership oracle with assumptions, thereby deviating from exact learning.

The third approach is a combination of passive and active techniques. In this approach, the target automaton is learned as complete as possible with the available information. When an unanswerable query is encountered, the query is either addressed at an external oracle, e.g. a domain expert, or translated into a test case which is executed against the SUT. Asking a domain expert leads to a guided learning approach. In this case, the learning is only semi-automatic. Executing a test case against an SUT could be conducted automatically. However, as the outcome of the query then would depend on the SUT, this approach compromises the independence of the learned automaton. As both approaches draw information from sources beside the test suite, inconsistencies could be introduced into the learning data.

7.4 Summary: A Suitable Approach With Room for Further Research

In this chapter, the design decisions of the approach to learning from test cases are revisited. The main issue is the size and coverage of the test suite used in the learning process. While the mapping of test cases to learning traces is intuitive and simple, the size of a test suite sufficient for learning can get inhibitively large. Optimizations to deal with this problem comprise the extension of the semantic state-merging approach to better exploit the information contained in the test cases and an extension of the learning algorithm to work with unanswerable membership queries. In addition, the relation between test suite coverage, system structure, and learnability offers interesting research topics.

8 Conclusion

In this thesis, a hybrid approach for learning from test cases has been presented. In the following, we summarize the results of the previous chapters and give an outlook on further work.

8.1 Summary

The hybrid learning approach combines both online and offline learning techniques to generate a deterministic finite automaton from a test suite. To control the learning process, Angluin's learning algorithm is used. This online learning algorithm works by querying assessments of traces from an oracle, which has been adapted to the characteristics of test cases. Offline state-merging methods are used to represent the test cases in a form that can be processed by the online learning algorithm. The properties of the test specification language are exploited to enable the recombination of test case traces. In summary, the hybrid approach combines a generic front-end with a test language specific back-end, thereby utilizing the advantages of both approaches.

The learning algorithm used as a front-end uses membership queries and equivalence queries to determine the target automaton. The membership queries are mapped to a search on the test suite, where a trace assessed as *pass* maps to an accepting trace and all other traces are mapped onto rejecting traces. An equivalence query is defined as a run of the test suite against the hypothesis automaton. It has been shown that while the membership queries need a test suite satisfying path coverage, for the equivalence query it is sufficient if the test suite satisfies a boundary-interior path coverage.

As a means to comply with the needs of both membership and equivalence queries, the trace graph has been introduced as an underlying data structure. Based on this data structure, the traces executed in an equivalence query can be limited in length, while at the same time providing longer traces for the membership queries. The trace graph satisfies the properties of a search tree, optimizing the search for a given trace and thereby the membership query. On the trace graph, a number of state-merging techniques have been defined, which exploit the features of the used test specification language to identify implicit traces and present them to the membership oracle of the learning algorithm.

The combined approach using Angluin's learning algorithm and the underlying trace graph structure has been implemented in a prototypical tool. Experiments show that while the combined approach performs better than Angluin's learning algorithm alone, the necessary size of the test suite still presents a limiting factor. In consequence, a number of possible alterations have been proposed that promise to amend the problematic points.

8.2 Outlook

Based on the analysis of the learning approach (Chapter 7), the next step is to incorporate the identified optimizations into the prototypical implementation. First and foremost, this comprises an extension of the state-merging methods. Initially, the merging will be based on the test language *Testing and Test Control Notation (TTCN-3)*. In parallel, existing approaches to learning parameterized models need to be adapted to the hybrid learning approach. As the inclusion of parameterized signals also affects the trace graph, the query oracle has to be adapted as well. Additionally, a re-implementation of the learning algorithm could be used as a means for optimization.

In the long run, the hybrid learning approach is intended as the basis of a monitoring framework. Based on an existing test suite, a model of the system under test is reconstructed via learning. The resulting model is then used as a reference oracle for monitoring the system under test. When a difference between the running system under test and the learned automaton is detected, the monitor notifies an administrator, presenting a trace reconstructing the error. Using the knowledge of the learning approach, this reconstructing trace could also be minimized, e.g. starting in a defined state and showing only signals on relevant ports. It could even be possible to correct the system under test automatically and test for regression by running the reconstructing trace against the corrected system under test. In this scenario, as the approach progresses from passive reconstruction of an automaton to the active testing of a system under test, the properties of the online learning approach will prove to be an advantage.

Another future application of the learning approach lies in test quality assessment. Inconsistencies between test cases or incompatibilities of test cases running on different interfaces could be detected during the learning process. In addition, by learning a model from an existing test suite and comparing it to the initial specification, the coverage of the test suite could be estimated. To this end, it is necessary to research the relations between system structure, test suite coverage, and learnability.

Bibliography

[AEY00] R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 304–313. ACM, 2000. Cited on page 6.

[AMN05] R. Alur, P. Madhusudan, and W. Nam. Symbolic Compositional Verification by Learning Assumptions. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005)*, volume 3576 of *Lecture Notes in Computer Science*, pages 548–562. Springer, 2005. Cited on page 8.

[Ang87] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987. Cited on pages 2, 8, 28, and 29.

[AO08] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. Cited on page 20.

[AOWZ09] S. Karimkhani Asl, C. Otto, S. Withus, and H. Zhang. Lernen von Automaten — Erweiterung. Project Report, University of Göttingen, 2009. Cited on pages 3 and 81.

[AS83] D. Angluin and C. H. Smith. Inductive Inference: Theory and Methods. *ACM Computing Surveys (CSUR)*, 15(3):237–269, 1983. Cited on page 6.

[Bau85] M. A. Bauer. Soundness and Completeness of a Synthesis Algorithm Based on Example Computations. *Journal of the ACM (JACM)*, 32(2):249–279, 1985. Cited on page 6.

[BBP75] A. W. Biermann, R. I. Baum, and F. E. Petry. Speeding up the Synthesis of Programs from Traces. *IEEE Transactions on Computers*, 24(2):122–136, 1975. Cited on page 6.

[BDGW97] J. L. Balcázar, J. Díaz, R. Gavaldà, and O. Watanabe. Algorithms for Learning Finite Automata from Queries: A Unified View. In D.-Z. Du and K.-I. Ko, editors, *Advances in Algorithms, Languages, and Complexity*, pages 53–72. Kluwer Academic Publishers, 1997. Cited on pages 6 and 104.

[BF72] A. W. Biermann and J. A. Feldman. On the Synthesis of Finite-State Machines from Samples of Their Behavior. *IEEE Transactions on Computers*, 21(6):592–597, 1972. Cited on page 27.

[BGJ$^+$05] T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the Correspondence Between Conformance Testing and Regular Inference. In *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005)*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2005. Cited on pages 57 and 105.

[BHKL09] B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-Style Learning of NFA. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 1004–1009, 2009. Cited on page 9.

[Bin99] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 1999. Cited on page 23.

[BJLS05] T. Berg, B. Jonsson, M. Leucker, and M. Saksena. Insights to Angluin's Learning. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 118:3–18, 2005. Cited on page 100.

[BJR06] T. Berg, B. Jonsson, and H. Raffelt. Regular Inference for State Machines with Parameters. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering (FASE 2006)*, volume 3922 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2006. Cited on pages 10, 100, and 104.

[BJR08] T. Berg, B. Jonsson, and H. Raffelt. Regular Inference for State Machines Using Domains with Equality Tests. In *Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE 2008)*, volume 4961 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2008. Cited on page 10.

[BK76] A. W. Biermann and R. Krishnaswamy. Constructing Programs from Example Computations. *IEEE Transactions on Software Engineering*, 2(3):141–153, 1976. Cited on pages 2 and 6.

[BKKL07] B. Bollig, J.-P. Katoen, C. Kern, and M. Leucker. Replaying Play In and Play Out: Synthesis of Design Models from Scenarios by Learning. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, volume 4424 of *Lecture Notes in Computer Science*, pages 435–450. Springer, 2007. Cited on page 9.

[BKKL08] B. Bollig, J.-P. Katoen, C. Kern, and M. Leucker. Smyle: A Tool for Synthesizing Distributed Models from Scenarios by Learning. In *Proceedings of the 19th International Conference on Concurrency Theory (CONCUR 2008)*, volume 5201 of *Lecture Notes in Computer Science*, pages 162–166. Springer, 2008. Cited on page 9.

[BO01] N. H. Bshouty and A. Owshanko. Learning Regular Sets with an Incomplete Membership Oracle. In *Proceedings of the 14th Annual Conference on Computational Learning Theory (COLT 2001) and 5th European Conference on Computational Learning Theory (EuroCOLT 2001)*, volume 2111 of *Lecture Notes in Computer Science*, pages 574–588. Springer, 2001. Cited on page 106.

[BRS$^+$00] L. Du Bousquet, S. Ramangalahy, S. Simon, C. Viho, A. F. E. Belinfante, and R. G. Vries. Formal Test Automation: The Conference protocol with TGV/Torx. In *Proceedings of the IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems: Tools and Techniques (TestCom 2000)*, IFIP Conference Proceedings, pages 221–228. Kluwer Academic Publishers, 2000. Cited on pages 81 and 83.

[CDLW04] J. E. Cook, Z. Du, C. Liu, and A. L. Wolf. Discovering models of behavior for concurrent workflows. *Computers in Industry*, 53(3):297–319, 2004. Cited on page 7.

[CF03] F. Coste and D. Fredouille. Unambiguous Automata Inference by Means of State-Merging Methods. In *Proceedings of the 14th European Conference on Machine Learning (ECML 2003)*, volume 2837 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 2003. Cited on page 7.

[CFC$^+$09] Y.-F. Chen, A. Farzan, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang. Learning Minimal Separating DFA's for Compositional Verification. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, volume 5505 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2009. Cited on page 8.

[CFKdlH04] F. Coste, D. Fredouille, C. Kermorvant, and C. de la Higuera. Introducing Domain and Typing Bias in Automata Inference. In *Proceedings of the 7th International Colloquium on Grammatical Inference: Algorithms and Applications (ICGI 2004)*, volume 3264 of *Lecture Notes in Computer Science*, pages 115–126. Springer, 2004. Cited on page 7.

[CM96] D. Carmel and S. Markovitch. Learning Models of Intelligent Agents. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 62–67. AAAI Press/MIT Press, 1996. Cited on pages 1 and 8.

[CO96] R. C. Carrasco and J. Oncina. Learning Deterministic Regular Grammars From Stochastic Samples in Polynomial Time. *Theoretical Informatics and Applications (RAIRO)*, 33(1):1–19, 1996. Cited on page 6.

[CW95] J. E. Cook and A. L. Wolf. Automating Process Discovery through Event-Data Analysis. In *Proceedings of the 17th International Conference on Software engineering (ICSE '95)*, pages 73–82. ACM, 1995. Cited on page 7.

[CW98] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(3):215–249, 1998. Cited on page 7.

[DBT06] R. Delamare, B. Baudry, and Y. Le Traon. Reverse-engineering of UML 2.0 Sequence Diagrams from Execution Traces. In *Workshop on Object-Oriented Reengineering at ECOOP 06*, 2006. Cited on page 7.

[DKU06] L. M. Duarte, J. Kramer, and S. Uchitel. Model Extraction Using Context Information. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, volume 4199 of *Lecture Notes in Computer Science*, pages 380–394. Springer, 2006. Cited on page 7.

[dlH05] C. de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, 2005. Cited on page 6.

[DR95] V. Diekert and G. Rozenberg, editors. *The Book of Traces.* World Scientific Publishing Co., 1995. Cited on page 11.

[ETS07] ETSI Standard (ES) 201 873 V3.2.1: The Testing and Test Control Notation version 3; Parts 1-9. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, also published as ITU-T Recommendation series Z.140, 2007. Cited on pages 20 and 64.

[FK01] B. Finkbeiner and I. Krüger. Using Message Sequence Charts for Component-Based Formal Verification. In *Proceedings of OOPSLA 2001 Workshop on Specification and Verification of Component-Based Systems*, 2001. Cited on page 6.

[FKR$^+$97] Y. Freund, M. J. Kearns, D. Ron, R. Rubinfeld, R. E. Schapire, and L. Sellie. Efficient Learning of Typical Finite Automata from Random Walks. *Information and Computation*, 138(1):23–48, 1997. Cited on page 8.

[FP98] N. E. Fenton and S. L. Pfleeger. *Software Metrics.* PWS Publishing Co., 1998. Cited on page 19.

[GCD06] N. D. Griffeth, Y. Cantor, and C. Djouvas. Testing a Network by Inferring Representative State Machines from Network Traces. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006)*, page 31. IEEE Computer Society, 2006. Cited on page 9.

[GdPAR08] P. García, M. V. de Parga, G. I. Álvarez, and J. Ruiz. Universal automata and NFA learning. *Theoretical Computer Science*, 407(1–3):192–202, 2008. Cited on page 7.

[GL06] O. Grinchtein and M. Leucker. Learning Finite-State Machines from Inexperienced Teachers. In *Proceedings of the 8th International Colloquium on Grammatical Inference: Algorithms and Applications (ICGI 2006)*, volume 4201 of *Lecture Notes in Computer Science*, pages 344–345. Springer, 2006. Cited on page 106.

[GLP06] O. Grinchtein, M. Leucker, and N. Piterman. Inferring Network Invariants Automatically. In *Proceedings of the Third International Joint Conference on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Computer Science*, pages 483–497. Springer, 2006. Cited on page 9.

[GLPS08] R. Groz, K. Li, A. Petrenko, and M. Shahbaz. Modular System Verification by Inference, Testing and Reachability Analysis. In *Proceedings of the 20th IFIP TC 6/WG 6.1 International Conference on Testing of Software and Communicating Systems (TestCom 2008)*, volume 5047 of *Lecture Notes in Computer Science*, pages 216–233. Springer, 2008. Cited on page 10.

[Gol67] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967. Cited on page 5.

[Har87] D. Harel. Statecharts: A Visual Formulation for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987. Cited on page 16.

[HBPU06] H. H. Hallal, S. Boroday, A. Petrenko, and A. Ulrich. A formal approach to property testing in causally consistent distributed traces. *Formal Aspects of Computing*, 18(1):63–83, 2006. Cited on page 7.

[HHNS02] A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model Generation by Moderated Regular Extrapolation. In *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002)*, volume 2306 of *Lecture Notes in Computer Science*, pages 80–95. Springer, 2002. Cited on page 9.

[HMU06] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 2006. Cited on pages 11 and 14.

[HNS03] H. Hungar, O. Niese, and B. Steffen. Domain-Specific Optimization in Automata Learning. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003)*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2003. Cited on pages 1 and 9.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991. Cited on pages 15, 16, 24, and 36.

[ITU99] ITU-T Recommendation Z.120: Message Sequence Charts (MSC). International Telecommunication Union (ITU-T), Geneve, 1999. Cited on page 6.

[KdlH02] C. Kermorvant and C. de la Higuera. Learning Languages with Help. In *Proceedings of the 6th International Colloquium on Grammatical Inference: Algorithms and Applications (ICGI 2002)*, volume 2484 of *Lecture Notes in Computer Science*, pages 161–173. Springer, 2002. Cited on page 7.

[KGSB98] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In *Proceedings of the IFIP WG10.3/WG10.5 International Workshop on Distributed and Parallel Embedded Systems (DIPES '98)*, volume 155 of *IFIP Conference Proceedings*, pages 61–72. Kluwer Academic Publishers, 1998. Cited on page 6.

[KM94] K. Koskimies and E. Mäkinen. Automatic synthesis of state machines from trace diagrams. *Software—Practice & Experience*, 24(7):643–658, 1994. Cited on page 6.

[KM03] I. H. Krüger and R. Mathew. Component Synthesis from Service Specifications. In *Revised Selected Papers of the International Workshop on Scenarios: Models, Transformations and Tools*, volume 3466 of *Lecture Notes in Computer Science*, pages 255–277. Springer, 2003. Cited on page 6.

[KV94] M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, 1994. Cited on pages 28, 30, 100, and 104.

[LDD08] B. Lambeau, C. Damas, and P. Dupont. State-Merging DFA Induction Algorithms with Mandatory Merge Constraints. In *Proceedings of the 9th International Colloquium on Grammatical Inference: Algorithms and Applications (ICGI 2008)*, volume 5278 of *Lecture Notes in Computer Science*, pages 139–153. Springer, 2008. Cited on pages 103 and 104.

[Leu07] M. Leucker. Learning meets Verification. In *Revised Lectures of the 5th International Symposium on Formal Methods for Components and Objects (FMCO 2006)*, volume 4709 of *Lecture Notes in Computer Science*, pages 127–151. Springer, 2007. Cited on page 6.

[LGS06] K. Li, R. Groz, and M. Shahbaz. Integration Testing of Distributed Components Based on Learning Parameterized I/O Models. In *Proceedings of the 26th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2006)*, volume 4229 of *Lecture Notes in Computer Science*, pages 436–450. Springer, 2006. Cited on pages 10, 100, and 104.

[LNS$^+$97] D. Lee, A. N. Netravali, K. K. Sabnani, B. Sugla, and A. John. Passive Testing and Applications to Network Management. In *Proceedings of the Fifth International Conference on Network Protocols (ICNP '97)*, page 113. IEEE Computer Society, 1997. Cited on page 24.

[Neu09] Dennis Neumann. Test Case Generation using Model Transformations. Master's thesis, University of Göttingen, Institute for Computer Science, Göttingen, Germany, 2009. Cited on page 82.

[NMA08] W. Nam, P. Madhusudan, and R. Alur. Automatic symbolic compositional verification by learning assumptions. *Formal Methods in System Design*, 32(3):207–234, 2008. Cited on page 8.

[NSV03] A. Netravali, K. Sabnani, and R. Viswanathan. Correct Passive Testing Algorithms and Complete Fault Coverage. In *Proceedings of the 23rd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2003)*, volume 2767 of *Lecture Notes in Computer Science*, pages 303–318. Springer, 2003. Cited on page 24.

[OG92] P. Oncina and J. García. *Advances in Structural and Syntactic Pattern Recognition*, volume 5 of *Series in Machine Perception and Artificial Intelligence*, chapter Identifying regular languages in polynomial time, pages 99–108. World Scientific Publishing, 1992. Cited on page 27.

[Ott09] Christian Otto. Refinement of Angluin's Machine Learning Algorithm to Learning Models from TTCN-3 Test Cases. Master's thesis, University of Göttingen, Institute for Computer Science, Göttingen, Germany, 2009. Cited on pages 4 and 81.

[PB76] F. E. Petry and A. W. Biermann. Reconstruction of algorithms from memory snapshots of their execution. In *Proceedings of the 1976 ACM Annual Conference/Annual Meeting*, pages 530–534. ACM, 1976. Cited on page 6.

[Pit89] L. Pitt. Inductive Inference, DFAs, and Computational Complexity. In *Proceedings of the International Workshop on Analogical and Inductive Inference (All '89)*, volume 397 of *Lecture Notes in Computer Science*, pages 18–44. Springer, 1989. Cited on page 6.

[Pol08] Sergei Polonski. Learning of Protocol-based Automata. Master's thesis, University of Göttingen, Institute for Computer Science, Göttingen, Germany, May 2008. Cited on page 3.

[RS93] R. L. Rivest and R. E. Schapire. Inference of Finite Automata Using Homing Sequences. *Information and Computation*, 103(2):299–347, 1993. Cited on page 8.

[RS94] R. L. Rivest and R. E. Schapire. Diversity-Based Inference of Finite Automata. *Journal of the ACM*, 41(3):555–589, 1994. Cited on page 8.

[SC07] N. Sinha and E. M. Clarke. SAT-Based Compositional Verification Using Lazy Learning. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*, volume 4590 of *Lecture Notes in Computer Science*, pages 39–54. Springer, 2007. Cited on page 8.

[SLG07] M. Shahbaz, K. Li, and R. Groz. Learning and Integration of Parameterized Components Through Testing. In *Proceedings of the 19th IFIP TC6/WG6.1 International Conference on Testing of Software and Communicating Systems (TestCom 2007)*, volume 4581 of *Lecture Notes in Computer Science*, pages 319–334. Springer, 2007. Cited on pages 1 and 10.

[SLS07] A. Spillner, T. Linz, and H. Schaefer. *Software Testing Foundations: A Study Guide for the Certified Tester Exam*. Dpunkt.Verlag, 2007. Cited on page 20.

[Som06] Ian Sommerville. *Software Engineering*. Addison Wesley, 8 edition, 2006. Cited on pages 16 and 19.

[ST94] R. H. Sloan and G Turán. Learning with queries but incomplete information (extended abstract). In *Proceedings of the seventh Annual Conference on Computational Learning Theory (COLT '94)*, pages 237–245. ACM, 1994. Cited on page 105.

[UBC07] S. Uchitel, G. Brunet, and M. Chechik. Behaviour Model Synthesis from Properties and Scenarios. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pages 34–43. IEEE Computer Society, 2007. Cited on page 6.

[UP07] A. Ulrich and A. Petrenko. Reverse Engineering Models from Traces to Validate Distributed Systems - An Industrial Case Study. In *Proceedings of the Third European Conference on Model Driven Architecture- Foundations and Applications (ECMDA-FA 2007)*, volume 4530 of *Lecture Notes in Computer Science*, pages 184–193. Springer, 2007. Cited on page 7.

[vdA09] W. M. P. van der Aalst. Process-Aware Information Systems: Lessons to Be Learned from Process Mining. *Transactions on Petri Nets and Other Models of Concurrency*, 2:1–26, 2009. Cited on page 7.

[vdAWM04] W. M. P. van der Aalst, T. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004. Cited on page 7.

[WGTZ08] E. Werner, J. Grabowski, S. Troschütz, and B. Zeiß. A TTCN-3-based Web Service Test Framework. In *Proceedings of Testing of Software - From Research to Practice 2008 (Workshop in Conjunction with the GI-conference Software Engineering 2008)*, volume 122 of *Lecture Notes in Informatics*, pages 375–382. Gesellschaft für Informatik, 2008. Cited on page 35.

[WNG04] E. Werner, H. Neukirchen, and J. Grabowski. Self-adaptive Functional Testing of Services Working in Continuously Changing Contexts. In *Proceedings of the 3rd Workshop on System Testing and Validation (SV04)*, Fraunhofer Book Series, 2004. Cited on page 3.

[WPG08] E. Werner, S. Polonski, and J. Grabowski. Using Learning Techniques to Generate System Models for Online Testing. In *INFORMATIK 2008, Beherrschbare Systeme - dank Informatik, Band 1, Proceedings of the 38. Annual Meeting of the Gesellschaft für Informatik e.V. (GI)*, volume 133 of *Lecture Notes in Informatics*, pages 183–186. Köllen Verlag, 2008. Cited on page 3.

Curriculum Vitae

Edith Benedicta Maria Werner

Persönliche Daten

Geb. am 14. Juli 1980 in Nürnberg

Staatsangehörigkeit: deutsch

Wissenschaftlicher Werdegang

10/2003–08/2009 Wissenschaftliche Mitarbeiterin am Institut für Informatik der Georg-August-Universität Göttingen

10/1998–06/2003 Studium der Informatik an der
Friedrich-Alexander-Universität Erlangen
Abschluss: Diplom

Schwerpunktfach: Kommunikationssysteme

Diplomarbeit mit dem Thema *Leistungsbewertung mit Multi-terminalen Binären Entscheidungsdiagrammen*
Note: 1.0

Studienarbeit mit dem Thema *Direkte Erzeugung symbolischer Darstellungen von Transitionssystemen aus Stochastischen Prozessalgebren*
Note: 1.0

09/1990–07/1998 Christian-Ernst-Gymnasium, Erlangen
Abschluss: Abitur

09/1986–07/1990 Grundschule Loschgeschule, Erlangen

Auszeichnungen

09/2004 Auszeichnung der Diplomarbeit durch den GI/ITG Fachausschusses "Messung, Modellierung und Bewertung von Rechensystemen"

04/2003 Auszeichnung der Studienarbeit durch das Institut für Informatik der Universität Erlangen-Nürnberg

Veröffentlichungen

[1] Edith Werner, Sergei Polonski, and Jens Grabowski. Using Learning Techniques to Generate System Models for Online Testing. In *INFORMATIK 2008, Beherrschbare Systeme - dank Informatik, Band 1, Proceedings of the 38. Annual Meeting of the Gesellschaft für Informatik e.V. (GI)*, volume 133 of *Lecture Notes in Informatics*, pages 183–186. Köllen Verlag, 2008.

[2] Edith Werner, Jens Grabowski, Stefan Troschütz, and Benjamin Zeiß. A TTCN-3-based Web Service Test Framework. In *Proceedings of Testing of Software — From Research to Practice 2008 (Workshop in conjunction with the GI-conference Software Engineering 2008)*, volume 122 of *Lecture Notes in Informatics*, pages 375–382. Gesellschaft für Informatik, 2008.

[3] Edith Werner, Jens Grabowski, Helmut Neukirchen, Nils Röttger, Stephan Waack, and Benjamin Zeiß. TTCN-3 Quality Engineering: Using Learning Techniques to Evaluate Metric Sets. In *Proceedings of 13th System Design Language Forum (SDL Forum 2007)*, volume 4745 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2007.

[4] Edith Werner, Helmut Neukirchen, and Jens Grabowski. Self-adaptive Functional Testing of Services Working in Continuously Changing Contexts. In *Proceedings of the 3rd Workshop on System Testing and Validation (SV04)*, Fraunhofer Book Series, 2004.

[5] Matthias Kuntz, Markus Siegle, and Edith Werner. Symbolic Performance and Dependability Evaluation with the Tool CASPA. In *Applying Formal Methods: Testing, Performance, and M/E-Commerce: Proceedings of the FORTE 2004 Workshops The FormEMC, EPEW, ITM*, volume 3236 of *Lecture Notes in Informatics*, pages 293–307. Springer, 2004.

[6] Edith Werner. Performance Evaluation using Multi-Terminal Binary Decision Diagrams. In *Proceedings of 12th GI/ITG Conference on Measuring and Evaluation of Computer and Communication Systems (MMB) together with 3rd Polish-German Teletraffic Symposium (PGTS) (MMB & PGTS 2004)*, pages 179–184. VDE Verlag, 2004.

[7] Matthias Kuntz, Markus Siegle, and Edith Werner. CASPA: A Tool for Symbolic Performance and Dependability Evaluation. In *Supplemental Volume of Proceedings of International Conference on Dependable Systems and Networks (DSN)*, pages 90–91, 2004.

[8] Matthias Kuntz, Markus Siegle, and Edith Werner. CASPA: A performance evaluation tool based on stochastic process algebra and symbolic data structures. In *Performance TOOLS 2003. Proceedings of the 13th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation.* Technical Report 781, University of Dortmund, 2003.