

Paper A

Therese Berg, and Harald Raffelt. Model Checking. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer Verlag, 2004. Reprinted with permission from Springer Verlag.

Manfred Broy
Bengt Jonsson
Joost-Pieter Katoen
Martin Leucker
Alexander Pretschner (Eds.)

Model-based Testing of Reactive Systems

A Seminar Volume

19 Model checking

Therese Berg¹ and Harald Raffelt²

¹ Computer Science Department
Uppsala University
email: thereseb@it.uu.se

² Chair of Programming Systems and Compiler Construction
University of Dortmund
email: harald.raffelt@cs.uni-dortmund.de

19.1 Introduction

When developing hard- or software systems one starts with a collection of requirements. Most requirements arise due to the needs of the customer, others originate from design decisions and further constraints. Of course, the final system should fulfill these requirements. Besides general requirements like scalability and performance, there is often a large number of formal requirements which concern the functionality of the system. Typical requirements of this kind are liveness requirements (e.g. bad things never happen), fairness requirements (e.g. the system continues doing meaningful things), and in general requirements which prescribe the chronological order of events (e.g. event A may only occur after event B).

A typical approach to meet this goal is to construct a model, to check that the model fulfills the requirements and then to show that the system conforms to the model. For the checking models against formal requirements automatical means have been developed in the past 20 years under the term *model checking*.

Previous chapters have presented various ways of testing. The first and the second part of book introduced some methods for testing finite state machines and labeled transition system. The third part was about model-based test generation. The general assumption of all approaches so far was, that the specification is completely given in form of a model. Then, testing becomes the task of checking whether a system conforms to a given specification or not. But in real life there is unfortunately in most cases no formal model of the system. To make use of, for example, conformance testing or methods for testing I/O-automata, it is often necessary to build the formal model by hand. This procedure is very error-prone, since in many cases one can only use the informal customer requirement specifications and some expert knowledge of the systems developer to build the model.

One solution to this problem comes from the area of *automata learning*, which provides some methods to generate a formal model out of a black box system. This approach allows at least to compare one version of the system with another one.

However, one can think of a more direct way to the requirements on the final system. After all, the main goal is usually to show that the system fulfills the

requirements and conformance to a hand-made model is just an instrument to achieve.

In this chapter we explain one possible way to check requirements on black boxes. It combines model checking with model learning. It is known under the terms *adaptive model checking* or *black box checking*. The idea is that a (part) of the model of the black box is learned by a machine learning algorithm. This model is then used for model checking the requirements. However, if a counter example is found, it might be because the system does not fulfill the requirement but, it can also be that the model is not adequate. In other words, the bug might be in the model not in the system. Then, the model has to be improved.

To make the chapter self-contained, we recall model checking techniques in the first part of the chapter (Section 19.3) and present learning algorithms in the second part (Section 19.4).

In the last part of the chapter (Section 19.5) the two techniques are combined.

19.2 Preliminaries

Definition 19.1. A **deterministic finite-state automaton (DFA)** is a 5-tuple $\mathcal{M} = (\Sigma, Q, \delta, q_0, F)$, where

- Σ is a finite set of **letters** called **alphabet**
- Q is a non-empty finite set of **states**
- $\delta : Q \times \Sigma \rightarrow Q$ is the **transition function**
- $q_0 \in Q$ is the **initial state**
- $F \subseteq Q$ is a set of **accepting states**

The machine starts in the initial state q_0 and reads a **string** or **word** of letters of its alphabet. It uses the transition function δ to determine the next state using the current state and the letter just read. Formally a word w is a sequence of letters $w = a_1 a_2 \dots a_n \in \Sigma^*$. The **empty word**, which has no letters, is usually denoted by ε . A **prefix** u of a word w is such that $w = uv$, where $w, u, v \in \Sigma^*$. The set of all finite words w with exactly n letters which can be build over an alphabet Σ is defined by $\Sigma^n = \varepsilon$ iff $n = 0$, and $\Sigma^n = \Sigma \cdot \Sigma^{n-1}$. The set of all finite words is denoted by Σ^* , which is defined by $\Sigma^* = \cup_{n \in \mathbb{N}} \Sigma^n$.

We denote the number of states Q , the size of the alphabet Σ , and the size of the transition function δ by respectively $|Q|$, $|\Sigma|$, and $|\delta|$. The latter is defined to be the number of elements of the domain of δ , i.e. $|Q \times \Sigma|$. Furthermore $q_i \xrightarrow{a} q_j$ is a denotation of $\delta(q_i, a) = q_j$.

Definition 19.2. Let $[n] = \{0, \dots, n\}$. A **finite run** π of a DFA \mathcal{M} on a finite word $w = a_0 a_1 \dots a_n \in \Sigma^*$ is a sequence of states $\pi = \pi_0 \dots \pi_{n+1}$, such that

- $\pi_0 = q_0$
- $\forall i \in [n] : \pi_i \xrightarrow{a_i} \pi_{i+1}$

The first state π_0 of the run is the initial state q_0 of \mathcal{M} and each next state π_{i+1} is reached by reading one letter a_i . A run is called **accepting**, if $\pi_{n+1} \in F$. The letters a_i read by a run form a $w = a_1 \dots a_n$. If the run is accepting, then the word read by the run is also said to be accepting. The language a DFA \mathcal{M} recognizes is denoted by $\mathcal{L}(\mathcal{M})$. It is defined as the set of accepting words. We call a language \mathcal{L} **regular** if there is a DFA accepting \mathcal{L} .

A different kind of automaton which operates on infinite words was introduced by Büchi [Büc62] for obtaining a decision procedure for the monadic second-order theory of structures with one successor. Later these automata were called Büchi automata. The main idea of Büchi automata is to operate on infinite input words $w = a_0 a_1 \dots \in \Sigma^\omega$, whereas Σ^ω denotes the set of all infinite words over the alphabet Σ .

Definition 19.3. A **Büchi automaton** is a 5-tuple $A = (\Sigma, Q, \delta, q_0, F)$, where

- Σ is a finite set of actions or letters,
- Q is a finite set of **states**,
- $\delta : Q \times \Sigma \rightarrow 2^Q$ is a **transition function**,
- $q_0 \in Q$ is a **initial state** and,
- $F \subseteq Q$ is a set of **accepting states**.

Starting from its initial state the automaton chooses nondeterministically a possible successor state in $\delta(q, a)$ of the current state q .

Definition 19.4. An **infinite run** π of a Büchi automaton A on a word $w = a_0 a_1 \dots \in \Sigma^\omega$ is a sequence $\pi = \pi_0 \pi_1 \dots \in Q^\omega$, such that

- $\pi_0 = q_0$
- $\pi_{i+1} \in \delta(\pi_i, a_i)$.

The first state π_0 of the run is the initial state q_0 of A and each next state π_{i+1} is one of the states reachable by reading one letter a_i . The states that occur infinitely many times in a run are $\text{inf}(\pi) = \{q \mid q \in Q \text{ and } q = \pi_i \text{ for infinitely many } i \geq 0\}$. An infinite run of a Büchi automaton is accepted if it visits accepting states infinitely often. Formally an infinite run $\pi = \pi_0 \pi_1 \pi_2 \dots$ is accepted iff $\text{inf}(\pi) \cap F \neq \emptyset$. An infinite word $w = a_0 a_1 \dots \in \Sigma^\omega$ is accepted by the automaton, if and only if there is an infinite run of the automaton which accepts the word. The language $\mathcal{L}(A)$ accepted by a Büchi automaton A is the set of all accepted words. The complement of a language $\mathcal{L}(A)$ accepted by a Büchi automaton is the set of all not accepted words, it is defined as $\overline{\mathcal{L}(A)} = \Sigma^\omega \setminus \mathcal{L}(A)$.

The length of a finite run $\pi = \pi_0 \pi_1 \dots \pi_n$ is the number of its elements denoted by $|\pi| = n + 1$. The length of an infinite run is denoted by $|\pi| = \infty$. For $0 \leq i < |\pi|$ the *suffix* of a run $\pi = \pi_0 \pi_1 \dots \pi_n$ starting with element π_i is denoted by $\pi^i = \pi_i \pi_{i+1} \dots$

19.3 Model Checking

In the last few years model checking has become a powerful and promising approach to automatic verification of systems. In general a model checker is a tool which checks whether a given structure M (called model) satisfies a certain logical constraint ϕ (called property). Typically models are represented by finite automata-like structures and properties are described in temporal logic. In contrast to conventional logics in temporal logics it is possible to describe temporal dependencies like one action must take place before another one. The model checker either confirms that the properties hold or reports that they are not satisfied by the model. Some model checkers can produce a path in the model which does not satisfy the property, a so called counterexample. Counterexamples can be understood as a reason for the unsatisfied property. Besides from providing models and properties no further user interaction is necessary for the entire model checking process. Because of its push-button approach model checking is a powerful verification tool even in large environments like hardware verification.

In Section 19.3.1 we give a brief introduction to models used to describe systems for model checking purposes and in Section 19.3.2 some common formalisms to describe properties of systems are provided. In Section 19.3.3 a common automata-theoretic model-checking algorithm is presented in detail.

19.3.1 Models

Model-checking typically depends on a discrete model of a system which describes the system behavior. Usually these models are graph structures where nodes represent the states of the system and edges represent transitions between the states. For model checking purposes these structures are typically finite, but model checking infinite structures is also possible [BCMS01]. These graphs without any further annotation are not expressive enough to provide an interesting description of the system. Two approaches are in common use: Kripke structures, where the nodes are annotated with so called atomic propositions, and labeled transition systems where the edges are annotated with so called actions. These two descriptions can be combined into so called Kripke transition systems [MOSS99].

In the following we present an introduction into Kripke structures.

Definition 19.5. A **Kripke structure (KS)** over a set AP of atomic propositions is a triple (S, R, I) , where

- S is a set of states,
- $R \subseteq S \times S$ is a transitions relation and,
- $I : S \rightarrow 2^{AP}$ is a labeling function

Each proposition describes a basic local property of the systems states. To each state of the system a set of atomic propositions is assigned by the labeling

function $I : S \rightarrow 2^{AP}$, describing which propositions are valid for that state. The labeling function is sometimes called interpretation.

A Kripke structure is called *total* if R is a total relation, otherwise it is called *partial*. A Kripke structure is called *rooted*, if one state $s_0 \in S$ is declared as initial state. For model checking purposes S and AP are usually finite.

Example 19.6. In Figure 19.1 a coffee percolator is modeled by a rooted Kripke structure. The set of atomic propositions is defined by $AP = \{coffee, coin\}$.

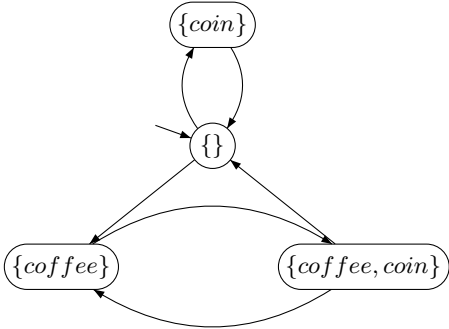


Fig. 19.1. Example Kripke structure

The atomic proposition *coffee* represents the fact that there is coffee-powder in the machine. In a state which is labeled by the atomic proposition *coffee*, the coffee-percolator is able to brew coffee and to spend it afterwards. The atomic proposition *coin* represents that there is a coin inside the coin slot and a user has paid for a coffee. Using this extra information one may imagine which actions lead from one state to another. To give a better understanding of the coffee percolator it is also represented as labeled transition system in the next example.

Example 19.7. In contrast to Kripke structures where the nodes are labeled with sets of atomic propositions labeled transition systems label the transitions with atomic actions. In Figure 19.2 the coffee percolator of the previous example is modeled as labeled transition system. Now one can see that, as long as there is no coffee-powder inserted, every inserted coin will be refused. Once coffee-powder is inserted every insert coin action will be answered by a spend coffee action until the automaton decides internally that one has to insert coffee-powder again.

19.3.2 Temporal Logics

Models describe systems, including their transitional behavior and local properties of the states. In order to model-check these systems, desired global characteristics of the system have to be formalized. For example, one might be interested in reachability properties like: "Is it possible to reach a state where

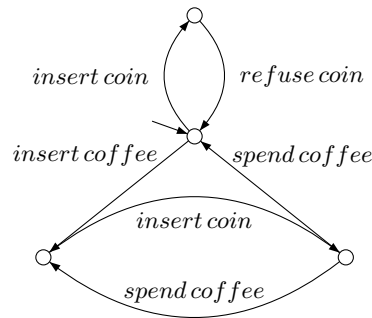


Fig. 19.2. Example Labeled Transition System

a certain atomic proposition holds, starting from the initial state?”. Temporal logics are logical formalisms designed for expressing such properties. There are two kinds of temporal logics, linear-time and branching-time. Linear-time logics are concerned with paths and treat each possible execution-path independently, branching-time logics, on the other hand, describe properties that depend on the branching structure of the model. The pros and cons of both logics are compared by Moshe Y. Vardi [Var01]. Both temporal logics have different expressiveness and therefore the kind of properties a model checker can prove depends on the choice of the underlying temporal logic. As an example, consider the two rooted labeled transitions systems in Figure 19.3, showing two different vending machines offering coffee and tea. Both machines serve coffee or tea after a coin has been inserted, but the right machine decides internally whether to serve coffee or tea, in contrast to the left machine which leaves the decision to the customer. Both machines have the same set of computations (maximal paths): $\{(coin, coffee), (coin, tea)\}$. Unfortunately they can not be distinguished in linear-time logics, since in linear-time logics each path is examined separately. Branching-time logic, in contrast, can distinguish these two machines, since it is possible to express properties like “a coffee action is possible after any coin action”.

The choice of using linear-time or branching-time logic depends on the properties to be analyzed. Linear-time logics are preferred when only path properties are of interest, as when analyzing data-flow properties, like dead-locks. Branching-time logics are often better for analyzing reactive systems, due to their greater selectivity.

Linear Temporal Logic (LTL)[Pnu77] can be seen as the “standard” linear-time logic. It is often presented in a form to be interpreted over Kripke structures. Its formulas are constructed as follows:

$$\phi ::= \text{true} \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{X}(\phi) \mid \phi \mathbf{U} \phi$$

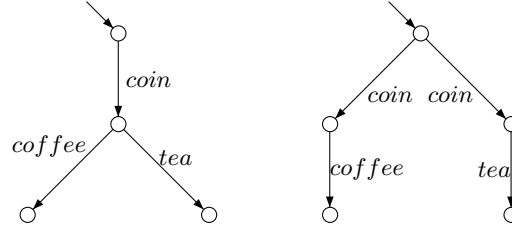


Fig. 19.3. Two vending machines

where p ranges over a set of atomic propositions AP . Note that sometimes **true** is defined to be a special atomic proposition, which is valid for every state.

The semantics $\llbracket \phi \rrbracket$ of a formula ϕ is the set of all runs π for which the property holds: $\llbracket \phi \rrbracket = \{\pi \mid \pi \models \phi\}$. The semantics is inductively defined on the structure of the formula.

$$\begin{aligned}
 \pi &\models \mathbf{true} \\
 \pi &\models p && \Leftrightarrow p \in I(\pi_0) \\
 \pi &\models \neg\phi && \Leftrightarrow \pi \not\models \phi \\
 \pi &\models \phi_1 \wedge \phi_2 && \Leftrightarrow \pi \models \phi_1 \wedge \pi \models \phi_2 \\
 \pi &\models \mathbf{X}(\phi) && \Leftrightarrow |\pi| > 1 \wedge \pi^1 \models \phi \\
 \pi &\models \phi_1 \mathbf{U} \phi_2 && \Leftrightarrow \exists k \in [|\pi| - 1] : (\pi^k \models \phi_2 \wedge \forall i \in [k - 1] : \pi^i \models \phi_1)
 \end{aligned}$$

Every run π satisfies **true** and every run π satisfies an atomic proposition, iff the first state π_0 of the run does. The negation and conjunction is interpreted as usual; further Boolean connectives may be introduced as abbreviations. E.g. $\phi_1 \vee \phi_2$, can be introduced as $\neg(\neg\phi_1 \wedge \neg\phi_2)$. The modality $\mathbf{X}(\phi)$ is called “next time ϕ ” and requires the property ϕ to hold for the next situation in the run. The modality $\phi_1 \mathbf{U} \phi_2$ is also denoted as $\mathbf{U}(\phi_1, \phi_2)$. It is called “ ϕ_1 until ϕ_2 ” and requires the property ϕ_1 to hold for all situations on the run until finally the property ϕ_2 holds for some situation.

Besides abbreviations of further Boolean connectivities, the following abbreviations are common:

$$\begin{aligned}
 \mathbf{F}(\phi) &:= \mathbf{U}(\mathbf{true}, \phi) \\
 \mathbf{G}(\phi) &:= \neg\mathbf{F}(\neg\phi)
 \end{aligned}$$

The modality $\mathbf{F}(\phi)$, called “finally ϕ ”, requires ϕ to hold for some later situation. The modality $\mathbf{G}(\phi)$, called “globally ϕ ”, requires ϕ to hold for all situations. The until modality $\phi_1 \mathbf{U} \phi_2$ is sometimes called **strong until** because it requires ϕ_2 to become true finally. In contrast to this modality, there is a different variant, called **weak until**, which holds, even if ϕ_2 never holds while ϕ_1 holds forever. ($\phi_1 \mathbf{WU} \phi_2 := \phi_1 \mathbf{U} \phi_2 \vee \mathbf{G}(\phi_1)$).

Since in system verification one is typically interested whether a specific state satisfies a certain property, there is the following convention: a state $s \in S$ of a transition system satisfies a formula if every run starting at s satisfies it.

Example 19.8. To illustrate the meaning of the modalities, here are some examples, that are satisfied by every run of the KS presented in Fig. 19.1:

- $X(\text{coffee} \vee \text{coin})$ states that in the second step of execution there will be coffee or a coin inside the automaton.
- $G(\text{coin} \rightarrow X(\neg \text{coin}))$ states that whenever there is a coin inside the automaton it will be removed in the next step.
- $G(\text{coffee} \rightarrow (\text{coffee} \text{WU} ((\text{coffee} \wedge \text{coin}) \rightarrow X\neg(\text{coffee} \vee \text{coin}))))$ states that once there is coffee inside the automaton it will last forever, or it will be removed in a later situation together with a coin in a single step.

Computational Tree Logic Computational Tree Logic (**CTL**) is one of the earliest proposed branching time logics. It can be considered as the branching time counterpart of LTL since it introduces universal and an existential path quantifiers. The syntax of CTL formulas is defined with respect to a KS (S, R, I) over a set AP of atomic propositions.

$$\phi ::= \text{true} \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \text{EX}(\phi) \mid \text{E}\phi\text{U}\phi \mid \text{A}\phi\text{U}\phi$$

As mentioned before p is an element of the atomic propositions AP . The semantics of a formula denote the subset of states $s \in S$ for which the formula holds: $\llbracket \phi \rrbracket^T = \{s \in S \mid s \models \phi\}$.

Let Π_s^* denotes the set of all runs starting with state s . The semantics of a formula ϕ is inductively defined on the structure of ϕ as follows:

$$\begin{aligned} s &\models \text{true} \\ s &\models p && \Leftrightarrow p \in I(s) \\ s &\models \neg\phi && \Leftrightarrow s \not\models \phi \\ s &\models \phi_1 \wedge \phi_2 && \Leftrightarrow s \models \phi_1 \wedge s \models \phi_2 \\ s &\models \text{EX}(\phi) && \Leftrightarrow \exists s' \in S : (s, s') \in R \wedge s' \models \phi \\ s &\models \text{E}\phi_1\text{U}\phi_2 && \Leftrightarrow \exists \pi \in \Pi_s^* : \exists k \in \llbracket |\pi| \rrbracket : (\pi^k \models \phi_2 \wedge \forall i \in [k-1] : \pi^i \models \phi_1) \\ s &\models \text{A}\phi_1\text{U}\phi_2 && \Leftrightarrow \forall \pi \in \Pi_s^* : \exists k \in \llbracket |\pi| \rrbracket : (\pi^k \models \phi_2 \wedge \forall i \in [k-1] : \pi^i \models \phi_1) \end{aligned}$$

Every state s satisfies **true** and every state s satisfies an atomic proposition, iff the proposition is assigned to the state. The negation and conjunction is interpreted as usual, further Boolean connectives may be introduced as abbreviations. The modality **EX** is called “exists next ϕ ”. It intuitive means that there is an immediate successor state s' reachable by executing one transition which satisfies ϕ . The modality **E** ϕ_1 **U** ϕ_2 , called “exists ϕ_1 until ϕ_2 ”, requires the existence of a run π starting with state s which has a prefix such that ϕ_2 holds for the last state of the prefix and ϕ_1 holds for all other states along the prefix. The modality **A** ϕ_1 **U** ϕ_2 is called “forall ϕ_1 until ϕ_2 ”. It requires that for every computation run π starting with state s , there is a prefix such that ϕ_2 holds for the last state of the prefix and ϕ_1 holds for all other states along the prefix. The following abbreviations are common:

$$\begin{aligned}
\text{AX}(\phi) &:= \neg\text{EX}(\neg\phi) \\
\text{AF}(\phi) &:= \text{AtrueU}\phi \\
\text{EF}(\phi) &:= \text{EtrueU}\phi \\
\text{AG}(\phi) &:= \neg\text{EF}(\neg\phi) \\
\text{EG}(\phi) &:= \neg\text{AF}(\neg\phi)
\end{aligned}$$

The modality $\text{AX}(\phi)$, called “forall next ϕ ”, requires that the property ϕ holds in every reachable successor state of s . The modality $\text{AF}(\phi)$, called “forall finally ϕ ”, requires that the property ϕ holds on every run (starting from the current state s) for some later state. The modality $\text{EF}(\phi)$ requires that there is a run starting from state s which satisfies the property ϕ for some later state. It is called “exists finally ϕ ”. The modality $\text{AG}(\phi)$, called “forall globally ϕ ”, requires that the property ϕ holds on every run for every state. The modality $\text{EG}(\phi)$ is called “exists globally ϕ ”. It requires that there is a run starting from state s which satisfies the property ϕ in every state.

Example 19.9. To illustrate the meaning of the modalities, here are some examples, which are satisfied by the KS presented in Figure 19.1:

- $\text{AX}(\text{coffee} \vee \text{coin})$ states that the automaton can make a step and then there will be coffee or a coin inside the automaton.
- $\text{AG}(\text{coin} \rightarrow \text{EX}(\neg\text{coin}))$ states that whenever there is a coin inside the automaton there is a next step which removes the coin.
- $\text{AG}(\text{AF}(\text{coin}))$ states that on every run a coin is infinite often inside the automaton.
- $\text{AG}(\text{coffee} \rightarrow (\text{EcoffeeU}((\text{coffee} \wedge \text{coin}) \rightarrow \text{EX}\neg(\text{coffee} \vee \text{coin}))))$ states that on every run once there is coffee inside the automaton it will last until it is finally removed in a later situation together with a coin in a single step.
- $\text{EG}((\text{coin} \rightarrow \text{EX}(\neg\text{coin})) \wedge (\neg\text{coin} \rightarrow \text{EX}(\text{coin})))$ states that there is a run of the coffee percolator where the condition coin and $\neg\text{coin}$ is alternating.

Other Temporal Logic Besides LTL and CTL there are a number of other temporal logics. Some of them extend the presented basic version of LTL and CTL to deal with special issues. **Fair computational tree logic** (FCTL) [EL85] for example extends CTL to deal with fairness constraints. Another well known extension to CTL is CTL*, which allows a more general combination of the universal and existential path quantifiers (A, E), and the until and next operator ($\text{X}(\phi), \phi\text{U}\phi$). In the following we shortly introduce two other temporal logics which are related to labeled transition systems.

- **Hennessy-Milner Logic (HML)** is a simple modal logic introduced by Hennessy and Milner [HM85, Mil89]. In contrast to LTL and CTL it is defined over a set of actions (Act) since it is related to labeled transition systems. HML is build out of the constant true, negation, conjunction, and the parameterized existential next operator

$\langle a \rangle$ ($a \in Act$). This modality is called “diamond $a \phi$ ” and holds if there is an a -transition to a state of the labeled transition system which satisfies ϕ . The important point about HML is that HML-properties can characterize finite automata up to bisimulation.

- **Modal μ -calculus**

was introduced by Kozen [Koz83] and extends Hennessy-Milner logic by a least fixpoint operator (μ). In general a fixpoint of a function f is a value x such that $f(x) = x$. Intuitively, the μ -calculus makes it possible to use modalities inside of recursively defined patterns. For example consider the CTL formula $EF(\phi)$. Another way of expressing this is to say that there is a property X such that either ϕ is satisfied in the current state or there is some successor state in which X is true. $X = \phi \vee \Diamond X$. This property can be expressed in μ -calculus as $\mu X. \phi \vee \Diamond X$.

Due to the extreme power of fixpoint operators the μ -calculus allows to express very complex properties within a sparse formalism. The μ -calculus covers LTL and CTL, and it is even possible to express fairness constraints which is not possible with the basic version of LTL and CTL.

19.3.3 Model Checking Algorithms

Model checking can be realized by several different approaches; prominent examples are the *semantic approach*, the *automata theoretic approach*, and the *tableau approach*.

The idea behind the semantic approach is to inductively compute the semantics of the formula in question to a given finite model, directly. This generates a set of states which satisfy the formula. The semantic approach is typically used for model checking branching time logics. There are efficient algorithms using this approach which operate linear in the size of the model even for the alternation free μ -calculus [CS92].

The automata theoretic approach is used for model checking linear-time logics and branching-time logics as well. This approach reduces the model checking problem to an inclusion problem between automata. An automaton A_ϕ is constructed from the property ϕ which accepts all runs satisfying ϕ . Another automaton A_M is constructed from model M which accepts the executions runs of the model. M satisfies ϕ if the language of the model-automaton A_M is a subset of language accepted by the properties automaton A_ϕ . This problem can be reduced to the problem of deciding non-emptiness of a product automaton which is possible by reachability analysis. As an example, an efficient algorithm for model checking LTL [Var96] is presented later.

The tableau method is used to determine if a certain state s of a given model M satisfies a property ϕ . This approach tries to construct a proof tree that witnesses that a given state satisfies a certain property. If no proof tree can be found, it provides a disproof (counterexample) of the property for the given state. Since the tableau method inspects only a small fraction of the state space [SW91], it combines well with incremental construction of the state space, which is a prominent approach to deal with the state explosion problem.

Another approach of fighting state explosion is to represent the transition relation of the models implicitly with an *ordered binary decision diagram* (OBDD) [BCMD90], since the size of the transition relation is the main limiting factor. By using common model checking algorithms with OBDDs and some refinements, very large examples with up to 10^{120} states have been verified [BCL92].

Model Checking LTL To model check Kripke structures with LTL-properties the following approach is proposed. In the first step the model M and the property ϕ are translated into automata models A_M and A_ϕ which represent the structures in a common way. The automaton A_M accepts all computations which are possible in the model and A_ϕ accepts all computations which are allowed with respect to the property. The model checking problem now reduces to the automata theoretic problem of checking that all computations accepted by an automaton A_M are also accepted by the automaton A_ϕ , that is $\mathcal{L}(A_M) \subseteq \mathcal{L}(A_\phi)$. Equivalently, one can check that the language $\mathcal{L}(A_M) \cap \overline{\mathcal{L}(A_\phi)}$ is empty. Instead of building the complement of the language accepted by A_ϕ it is possible to use the language of the complement automaton $\overline{A_\phi}$, which is defined such that it accepts the words of the complement language $\mathcal{L}(\overline{A_\phi}) = \overline{\mathcal{L}(A_\phi)}$. Complement automata were first studied by Büchi [Büc62], a definition and construction in the context of temporal logics is given by Sistla, Vardi, and Wolper [SVW87].

Since A_ϕ exactly accepts the computations satisfying ϕ the negation $\mathcal{L}(\overline{A_\phi})$ of the automaton can be expressed by negation of the property. $\overline{A_\phi} = A_{\neg\phi}$

There is a number of approaches how to transform an LTL property into an automaton. One basic approach presented in the following model checking algorithm was purposed by Wolper, Vardi and Sistla in 1983 [WVS83], but there are some improved versions. Gastin and Oddoux for example present in “Fast LTL to Büchi Automata Translation” [GO01] a different method which use a variation of Büchi automata (very weak alternating automata) as intermediate step. Etesami and Holzmann suppose a method for “Optimizing Büchi Automata” [EH00] to reduce the size of the automata.

The following basic LTL model checking algorithm presented by Moshe Y. Vardi in 1996 [Var96] is structured in 5 steps:

- (1) The Kripke structure M , which represents the model, is translated into a Büchi automaton.
- (2) The LTL-property ϕ is translated into an alternating Büchi automaton $\mathcal{A}_{\neg\phi}$ which exactly accepts the computations satisfying $\neg\phi$. (Alternating Büchi automaton are introduced in the later Definition 19.11)
- (3) The alternating Büchi automaton $\mathcal{A}_{\neg\phi}$ is translated into a nondeterministic Büchi automaton $A_{\neg\phi}$ which exactly accepts the same set of computations.
- (4) The language intersection of A_M and $A_{\neg\phi}$ is build, such that $\mathcal{L}(A_M \cap A_{\neg\phi}) = \mathcal{L}(A_M) \cap \mathcal{L}(A_{\neg\phi})$
- (5) The language $\mathcal{L}(A_M \cap A_{\neg\phi})$ is checked for emptiness.

If $\mathcal{L}(A_M \cap A_{\neg\phi})$ is empty then $M \models \phi$. On the other hand, if $\mathcal{L}(A_M \cap A_{\neg\phi})$ is not empty there is at least one run of $A_M \cap A_{\neg\phi}$ which is accepted by the model M

but not by the property ϕ . This run can be used as a counterexample, giving a reason why the model does not satisfy the property. In the following we explain each step of the algorithm in detail.

Step 1: The Kripke structure M , which represents the model, is translated into a Büchi automaton.

A rooted Kripke structures $M = (S, s_0, R, I)$ over a set of atomic propositions AP can be viewed as a Büchi automaton $A_M = (\Sigma, Q, \delta, q_0, F)$ where the set of states are equal $Q = S$, the initial states are equal $q_0 = s_0$, every state of the Büchi automaton is accepting $F = Q$, each action is a subset of atomic propositions $\Sigma = 2^{AP}$ and the transition function is defined as follows:

$$\delta : (q, a) \in Q \times \Sigma \mapsto \{q' \mid (q, q') \in R \wedge a = I(q)\}$$

Example 19.10. In Figure 19.4 the Büchi automaton constructed from the KS in Figure 19.1 is shown.

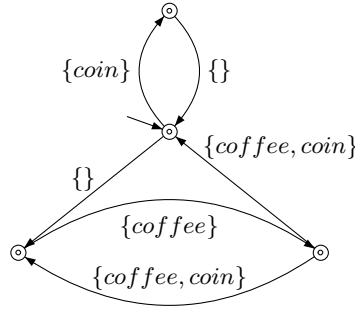


Fig. 19.4. Büchi automaton example

Step 2: The LTL-Property ϕ is translated into an alternating Büchi automaton $\mathcal{A}_{\neg\phi}$ which exactly accepts the computations satisfying $\neg\phi$.

Nondeterminism of (Büchi automata) can be understood as a kind of existential choice; a successor state s' of a state s is one of the states s' in $\delta(s)$. The dual of existential choice is universal choice, and therefore it is natural to consider automata that have the power of existential choice and universal choice. Such automata are called alternating. An alternating Büchi automaton is defined with respect to a set of positive Boolean formulas $\mathcal{B}^+(D)$. Positive Boolean formulas ϕ_B over a set D of variables are constructed as follows where $d \in D$:

$$\phi_B := d \mid \phi_B \dot{\vee} \phi_B \mid \phi_B \dot{\wedge} \phi_B \mid \text{true} \mid \text{false}$$

Note that the subscript point of the conjunction $\dot{\wedge}$ and disjunction $\dot{\vee}$ of positive Boolean formulas is used to differentiate them from conjunction \wedge and disjunction \vee of LTL formulas. Consider a nondeterministic Büchi automaton which

has a transition function including $\delta(q_0, a) = \{q_1, q_2, q_3\}$. This mapping can be written as $\delta(q_0, a) = q_1 \vee q_2 \vee q_3$ using positive Boolean formulas. In an alternating Büchi automaton one can have mappings like $\delta(q_0, a) = (q_1 \wedge q_2) \vee (q_3 \wedge q_4)$, meaning that the automaton starts from its initial state q_0 with an a -transition and can continue in both states q_1, q_2 or in both states q_3, q_4 . Note that an alternating Büchi automaton can continue in more than one state at the same time.

Definition 19.11. An **alternating Büchi automaton** $(\Sigma, Q, \delta, q_0, F)$ is a Büchi automaton where the transition function is defined as follows:

$$\delta : Q \times \Sigma \rightarrow \mathcal{B}^+(Q)$$

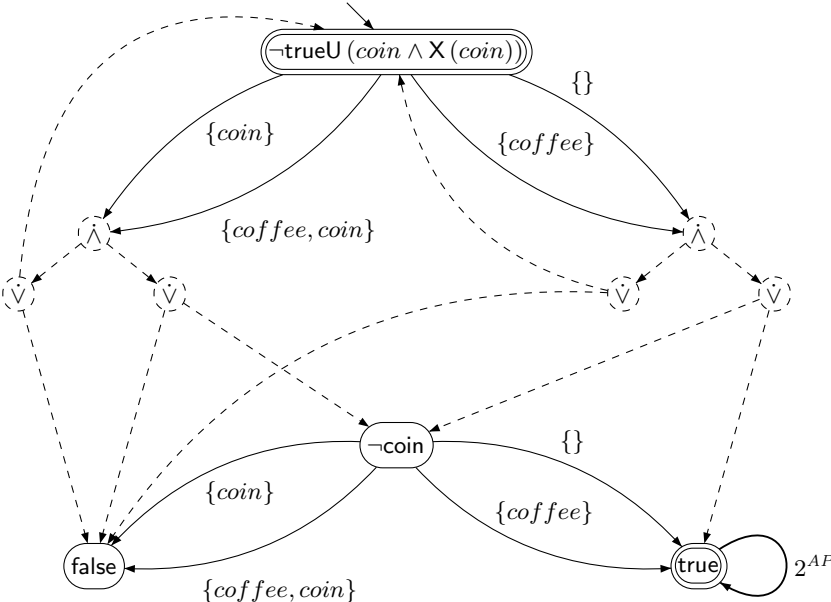


Fig. 19.5. Example Alternating Büchi Automaton

In Figure 19.5 a graphical representation of an alternating Büchi automaton is presented, visualizing the transition function of an automaton in an abstract way. The dotted lines are used to identify edges which belong to positive Boolean combination of states.

Because of the universal choice in alternating transitions, a run of an alternating automaton is a tree rather than a sequence. A *tree* $R = (r, p)$ is an infinite directed acyclic graph where r is a set of nodes and p is a *parent* function. One

node designated as the root, denoted by $\varepsilon \in r$. The root ε has no parent and every other node $n \neq \varepsilon$ has a unique parent. The *children* of a node $c(n)$ are the nodes n' which have n as parent. $c(n) := \{n' \mid n = p(n')\}$. The *level* $|n|$ of a node n is the distance from the root ε to the node: the root's level is $|\varepsilon| = 0$ and $|n| = 1 + |p(n)|$. A *branch* $\beta = n_0 n_1 \dots$ of a tree is a infinite sequence of nodes such that n_0 is the root ε and for all other nodes n_i ($i > 0$) of the branch the predecessor of a node in the branch is its parent: $n_{i-1} = p(n_i)$.

Definition 19.12. A run of an alternating Büchi automaton on a word $w = a_1 a_2 \dots \Sigma^\omega$ is a state-labeled tree (R, L) , where R is a tree and L is a mapping from the nodes of the tree r to the states, such that $r(\varepsilon) = q_0$ and the following holds:

- Each node n with level $|n| = i \mid \pi \mid$ of the tree r has k children n_1, \dots, n_k such that $\{L(n_1), \dots, L(n_k)\}$ satisfies $\delta(L(n), a_i)$

Note that the maximal level of a node in R is at most $\mid \pi \mid$. Not all branches need to reach such depth, since if $\delta(L(n), a) = \text{true}$, then n does not need to have any children. On the other hand we can not have $\delta(L(n), a) = \text{false}$, since false is not satisfiable.

For an alternating Büchi automaton a run (r, L) is accepting, iff every infinite branch visits accepting states infinitely often. Note that **true** and **false** are special states. For any action both states have only a single transition to itself. The state **true** is accepting and the state **false** is not accepting. Therefore a run with a branch visiting a **false**-state can not be accepting and a run with a branch visiting a **true**-state is accepting if all other branches visit accepting states infinitely often. The language $\mathcal{L}(\mathcal{A})$ of an alternating Büchi automaton \mathcal{A} is determined by all words for which an accepting run exists. Note that for a word w there may be more than one accepting run.

Example 19.13. Figure 19.6 outlines a run of the Büchi automaton of Figure 19.5 on the infinite word $w = (\{\text{coin}\}\}^\omega$.

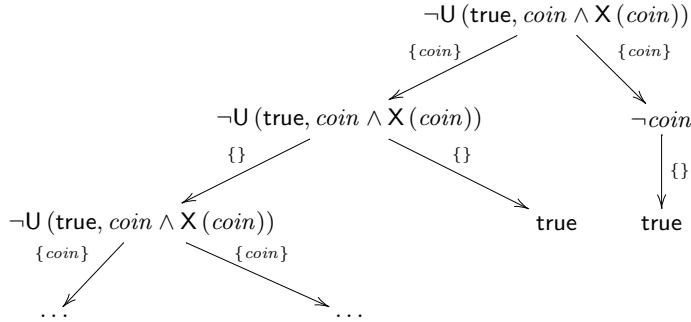


Fig. 19.6. Example run of an alternating Büchi automaton

The alternating Büchi automaton we are going to construct from a property ϕ uses the set of all sub-formulas and their negations as the set of states.

Definition 19.14. The set of sub-formulas $Sub(\phi)$ of a property ϕ is inductively defined on the structure of ϕ by:

$$\begin{aligned} Sub(\text{true}) &= \{\text{true}\} \\ Sub(\neg\phi) &= \{\neg\phi\} \cup Sub(\phi) \\ Sub(\mathbf{X}(\phi)) &= \{\mathbf{X}(\phi)\} \cup Sub(\phi) \\ Sub(\phi_1 \vee \phi_2) &= \{\phi_1 \vee \phi_2\} \cup Sub(\phi_1) \cup Sub(\phi_2) \\ Sub(\mathbf{U}(\phi_1, \phi_2)) &= \{\mathbf{U}(\phi_1, \phi_2)\} \cup Sub(\phi_1) \cup Sub(\phi_2) \end{aligned}$$

The transition function of an alternating Büchi automaton maps states to *positive* Boolean combinations of states. Since properties in LTL may use the negation modality $\neg\phi$ and negation is not allowed in positive Boolean functions, negation of properties is expressed by negation of states. For this reason the negatives of the sub-formulas are included into the set of states of the alternating Büchi automaton. To connect the negation of properties to the negation of positive Boolean combinations of states the following construction is used:

Definition 19.15. The dual $\overline{\phi}$ of a positive Boolean formula is defined inductively on the structure of a formula ϕ as follows:

$$\begin{aligned} \overline{\text{true}} &= \text{false} \\ \overline{\text{false}} &= \text{true} \\ \overline{\neg\phi} &= \phi \\ \overline{\phi_1 \vee \phi_2} &= \overline{\phi_1} \wedge \overline{\phi_2} \\ \overline{\phi_1 \wedge \phi_2} &= \overline{\phi_1} \vee \overline{\phi_2} \\ \overline{\mathbf{X}(\phi)} &= \neg\mathbf{X}(\phi) \\ \overline{\mathbf{U}(\phi_1, \phi_2)} &= \neg\mathbf{U}(\phi_1, \phi_2) \end{aligned}$$

Given an LTL formula ϕ , one can directly build an alternating Büchi automaton $A_\phi = (\Sigma, Q, \delta, q_0, F)$, such that $\mathcal{L}(A_\phi)$ is exactly the set of computations satisfying the property ϕ . The set of states Q is defined as the set of sub-formulas of ϕ and their negations. The set of actions is defined as $\Sigma = 2^{AP}$. The set of accepting states F consists of all formulas ϕ which have got the form $\neg\mathbf{U}(\phi_1, \phi_2)$. The transition function δ is inductively defined on the structure of ϕ as follows:

$$\begin{aligned} \delta(p, A) &= \begin{cases} \text{true} & \text{if } p \in A \\ \text{false} & \text{if } p \notin A \end{cases} \\ \delta(\phi_1 \vee \phi_2, A) &= \delta(\phi_1, A) \dot{\vee} \delta(\phi_2, A) \\ \delta(\phi_1 \wedge \phi_2, A) &= \delta(\phi_1, A) \dot{\wedge} \delta(\phi_2, A) \\ \delta(\neg\phi, A) &= \overline{\delta(\phi, A)} \\ \delta(\mathbf{X}(\phi), A) &= \phi \\ \delta(\mathbf{U}(\phi_1, \phi_2), A) &= \delta(\phi_2, A) \dot{\vee} (\delta(\phi_1, A) \dot{\wedge} \mathbf{U}(\phi_1, \phi_2), A) \end{aligned}$$

The idea behind this recursive definition is: whenever a composed formula is to check it is transformed into a Boolean combination of new formulas. In this

way a goal is reduced to several subgoals like in an tableau construction. Since any formula except the one of type $\text{U}(\phi_1, \phi_2)$ is turned into smaller sub-formulas, every finite branch of a potential run reaches either **true** or **false**. Every infinite branch has to hit states of the type $\text{U}(\phi_1, \phi_2)$ or $\neg\text{U}(\phi_1, \phi_2)$ infinitely often. If $\text{U}(\phi_1, \phi_2)$ is hit infinitely often it means, that the automaton fails to show that $\text{U}(\phi_1, \phi_2)$ holds and ϕ_2 is not satisfied on this branch. Therefore the state $\text{U}(\phi_1, \phi_2)$ is not included into the set of accepting states. On the other hand, if $\neg\text{U}(\phi_1, \phi_2)$ is hit infinitely often on a branch the automaton is not able to show ϕ_1 or ϕ_2 and $\neg\text{U}(\phi_1, \phi_2)$ holds. That is the reason for putting the state of type $\neg\text{U}(\phi_1, \phi_2)$ into the set of accepting states.

Example 19.16. Consider the property $\phi = \text{G}(\text{coin} \rightarrow \text{X}(\neg\text{coin}))$. In the following the construction of the alternating Büchi automaton \mathcal{A}_ϕ is presented. The underlying Kripke structure has got $AP = \{\text{coin}, \text{coffee}\}$ as the set of atomic propositions and therefore the set of actions is:

$$\Sigma = \{\{\}, \{\text{coin}\}, \{\text{coffee}\}, \{\text{coin}, \text{coffee}\}\}.$$

Since it is obvious how to get the set of states Q , it remains to calculate the transition function δ . It is easy to see that $\phi = \text{G}(\text{coin} \rightarrow \text{X}(\neg\text{coin}))$ is equivalent to $\neg\text{U}(\text{true}, \text{coin} \vee \text{X}(\text{coin}))$. Using the definition of the duality and the recursive definition of δ we get:

$$\begin{aligned} & \delta(\neg\text{U}(\text{true}, \text{coin} \wedge \text{X}(\text{coin})), \{\text{coin}\}) \\ = & \overline{\delta(\text{U}(\text{true}, \text{coin} \wedge \text{X}(\text{coin})), \{\text{coin}\})} \\ = & \overline{\delta(\text{coin} \wedge \text{X}(\text{coin}), \{\text{coin}\}) \vee (\delta(\text{true}, \{\text{coin}\}) \wedge \text{U}(\text{true}, \text{coin} \wedge \text{X}(\text{coin})))} \\ = & \overline{\delta(\text{coin} \wedge \text{X}(\text{coin}), \{\text{coin}\}) \wedge (\delta(\text{true}, \{\text{coin}\}) \wedge \text{U}(\text{true}, \text{coin} \wedge \text{X}(\text{coin})))} \\ = & \overline{\delta(\text{coin}, \{\text{coin}\}) \wedge \delta(\text{X}(\text{coin}), \{\text{coin}\})} \wedge \left(\overline{\delta(\text{true}, \{\text{coin}\})} \vee \text{U}(\text{true}, \text{coin} \wedge \text{X}(\text{coin})) \right) \\ = & \left(\overline{\delta(\text{coin}, \{\text{coin}\})} \vee \overline{\delta(\text{X}(\text{coin}), \{\text{coin}\})} \right) \wedge (\overline{\text{true}} \vee \neg\text{U}(\text{true}, \text{coin} \wedge \text{X}(\text{coin}))) \\ = & (\overline{\text{true}} \vee \overline{\text{coin}}) \wedge (\text{false} \vee \neg\text{U}(\text{true}, \text{coin} \wedge \text{X}(\text{coin}))) \\ = & (\text{false} \vee \neg\text{coin}) \wedge (\text{false} \vee \neg\text{U}(\text{true}, \text{coin} \wedge \text{X}(\text{coin}))) \end{aligned}$$

If one calculates the transition function for each (reachable) state and each set of atomic propositions, one gets the alternating Büchi automaton shown in Figure 19.5. Note that some edges have been joined in the graph because of the symmetry of $\{\}$ and $\{\text{coffee}\}$ respectively $\{\text{coin}\}$ and $\{\text{coin}, \text{coffee}\}$.

Step 3: The alternating Büchi automaton $\mathcal{A}_{-\phi}$ is translated into a nondeterministic Büchi automaton $A_{-\phi}$ which exactly accepts the same set of computations.

Two problems arise during the transformation of alternating Büchi automaton $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ into nondeterministic Büchi automaton $A' = (\Sigma, Q', \delta', q'_0, F')$: How to deal with the universal choice and which states should be accepting?

To differentiate states of the alternating Büchi automaton and states of the nondeterministic Büchi automaton we call them alternating states and nondeterministic states.

Obviously the conjunction of states in alternating Büchi automata \mathcal{A} is not directly transferable to nondeterministic Büchi automata A . The solution to this problem is similar to the power-set construction mapping nondeterminism to determinism. If each state of the nondeterministic automaton A consist of a set of states of \mathcal{A} one can translate transitions with conjunction as follows. Consider a transition $\delta(q, a) = q_1 \wedge q_2 \vee q_3 \wedge q_4$. Using a power-set construction we can map this transition to a nondeterministic transition $\delta'(\{q\}, a) = \{\{q_1, q_2\}, \{q_3, q_4\}\}$. The idea of this construction is to map the states $\{q\}$ to sets of minimal sets satisfying the transitions positive Boolean condition. If the starting state of a transitions consists of more than one state the transition maps to minimal sets of states which satisfy the conjunction of all transition conditions. Let $U, X \in Q'$ subsets of alternating states. Formally we map $\delta'(U, a)$ to a new state X such that $X \models \bigwedge_{q \in U} \delta(q, a)$.

This construction is not sufficient to define the accepting states correctly. Surely a set of alternating states has to be accepting if all its states are accepting, but accepting states of alternating Büchi automata do not have to occur in positive Boolean combinations at the same time. A run of an alternating Büchi automaton is accepted if each infinite branch of the run hits accepting states infinitely often, but the accepting states can occur on different levels of each branch of the run. In other words; the (alternating) accepting states of an accepting run can occur one after another on a run of the nondeterministic automaton. The nondeterministic Büchi automaton has to collect the accepting states it has visited. Therefore we define the states for the nondeterministic Büchi automaton as follows: $Q' = 2^Q \times 2^Q$. The first component of this tuple contains non accepting states for which no accepting state was seen recently. The second component is used to collect accepting states and states for which accepting states have been visited. The idea is that successor states of accepting states are shifted from the first component to the second. Thus, the empty set in the first component identifies that for all alternating states of the current (nondeterministic) state accepting states have been visited and therefore we define the set of accepting states as $F' = 2^\emptyset \times 2^Q$.

If the initial state q_0 is not accepting and we have not seen any accepting state initially we define the initial state $q'_0 = (q_0, \emptyset)$. If q_0 is an accepting state we define $q'_0 = (\emptyset, q_0)$.

For a pair $(U, V) \in Q'$ and an action a let δ' yield the pairs $(U', V') \in Q'$ defined as follows:

- case $U \neq \emptyset$: Let $X, Y \subseteq Q$ be minimal sets satisfying the transitions requested by the states of respectively U and V reading the input symbol then $X \models \bigwedge_{q \in U} \delta(q, a)$ and $Y \models \bigwedge_{q \in V} \delta(q, a)$. We put non-accepting states in the first component and the accepting states in the second component. Furthermore, we add all members of Y to the second component except the ones which are also in the first component: $U' = Y - F$ and $V' = (X \cap F) \cup (Y - U')$.
- case $U = \emptyset$: Let $Y \subseteq Q$ a minimal set such that $Y \models \bigwedge_{q \in V} \delta(q, a)$. Since for all states in U we have seen an accepting state we are going to restart

collecting accepting states. Therefore we put all states into U' except the ones which are accepting states. $U' = Y - F$ and $V' = Y \cap F$.

Note that if the minimal set satisfying a transition is empty, the transition is always satisfied and therefore it is identified with the state **true**.

Step 4: The intersection of A_M and $A_{\neg\phi}$ is build, such that $\mathcal{L}(A_M \cap A_{\neg\phi}) = \mathcal{L}(A_M) \cap \mathcal{L}(A_{\neg\phi})$

Let $A' = (\Sigma, Q', \delta', q'_0, F')$ and $A'' = (\Sigma, Q'', \delta'', q''_0, F'')$ be two nondeterministic Büchi automata. We can build an automaton $A = A' \cap A''$ that accepts $\mathcal{L}(A') \cap \mathcal{L}(A'')$ as: $A = (\Sigma, Q' \times Q'' \times \{0, 1, 2\}, \delta, (q'_0, q''_0, 0), Q' \times Q'' \times \{2\})$. The transition function δ is defined as $\delta((q', q'', x), a) = \{(r', r'', y)\}$ such that both automata read each input symbol simultaneously $\delta'(q', a) = r'$ and $\delta''(q'', a) = r''$ and the third element of the state tuple, counting which automata has visited an accepting state is set as follows:

$$y = \begin{cases} 0 & x = 2 \\ 1 & x = 0 \wedge r' \in F' \\ 2 & x = 1 \wedge r'' \in F'' \\ x & \text{else} \end{cases}$$

Since accepting states of both automata may not appear together even if they appear individually infinitely often the setting $F = F' \times F''$ does not work. Therefore the third component is used to ensure that there is an accepting state if and only if both automata have visited an accepting state. The third component is initially 0 meaning that no automaton has visited an accepting state. It changes from 0 to 1 if the first automaton has seen an accepting state and it changes from 1 to 2 if the second automaton has also visited an accepting state. If both states have visited accepting states then there is an accepting state in $A' \cup A''$ and the search for accepting states is restarted with setting the third component back to 0

Step 5: Decide if the intersection of A_M and $A_{\neg\phi}$ is empty.

Since the number of accepting states of a Büchi automaton is finite infinite accepting runs have to visit single accepting states infinitely often. Therefore if an accepting run of a Büchi automaton exists there has to be a cycle in the graph of the automaton which is reachable from the initial state. It is a well known fact that using a depth-first-search algorithm one can search the graph of a Büchi automaton for a reachable cycle in linear time with respect to the number of nodes plus the number of edges. If the states of these cycles intersect with the set of accepting states the language of the automaton is not empty.

19.3.4 Model Checking Tools

In the last few years model checking has become a powerful and promising approach to automatic verification of systems. In order to be suitable for different

purposes there are a number of model checking algorithms which work on different types of models and temporal logics. Model checking has become a common technique since in the last two decades a number of model checking tools have been developed.

The first one was *COSPAN* [HK87, HHK96] which has been in use (and continuous development) since 1986. It has been applied to a number of commercial projects, as well as having been licensed to numerous universities for educational use.

Another model checking tool is *Murphi* ($\text{Mur}\varphi$). It focuses on protocol verification and its specification facilities are limited, since it is not possible to define properties of sequential behavior. It is only possible to detect deadlocks, predefined error-states and states that violate a kind of Boolean invariant.

There are two more well-known tools which deal with process specifications written in the verification languages like *Promela* (a Process Meta Language) and *LOTOS*. The model checker *SPIN* [Hol97] is a generic verification system that supported design and verification of asynchronous process systems. It focuses on providing the correctness of process interactions. *SPIN* accepts models that are described in *Promela* and properties specified in the syntax of linear time logic. *SPIN* uses an automata-theoretic approach with on-the-fly construction of the automata. Another model checking tool is *OPEN/CAESAR* [FGM⁺92]. It was the first model checking tool that supports the standardized process specification language *LOTOS*, but *OPEN/CAESAR* has a generic API to support other process descriptions as well. *OPEN/CAESAR* supports the modal μ -calculus and uses an automata theoretic approach with on-the-fly construction as well to fight the state explosion problem.

In contrast to *Murphi* and *SPIN* the *Fixpoint-Analysis Machine* [SCK⁺95] and the *Concurrency Workbench* [CPS93] are designed to support a wide range of applications. The *Fixpoint-Analysis Machine* can deal with the modal μ -calculus. The tool works not only on labeled transition systems but also on context-free processes (i.e. processes that are given in terms of a context-free grammar). The *Concurrency Workbench* is designed to incorporate several different verification methods in a modular fashion. As well as the *Fixpoint-Analysis Machine* it supports the modal μ -calculus.

The model checker *SMV* (Symbolic Model Verifier) was designed to deal with the state explosion problem. It uses an OBDD-based (Ordered Binary Decision Diagram) algorithm and supports properties specified in CTL. It has some extensions to verify fairness constraints. A new variant of *SMV* is the *NuSMV* [CCG⁺02] project which aims at the development of a state-of-the-art symbolic model checker, designed to be applicable in technology transfer. It is based on *SMV* and uses essentially the same input language as *SMV*. The main novelty in this open source project is the integration of model checking techniques based on propositional satisfiability.

19.4 Learning Finite State Machines

Techniques such as model checking and model-based test-generation are a convenient way to automatically improve a system’s reliability as a system conforming to its specification. The problem is that in many cases a model of the system does not exist or if it does, it is outdated. If the model is to be constructed by hand this may be time-consuming and it is very much dependent on how familiar the test engineer is with the system under test (SUT). A way to facilitate the test engineer’s work and derive a more reliable model, is to automate the generation of a model of the SUT. A proposed procedure to attain this is to apply a technique called *model learning*, sometimes also called model inference.

This section will explain how a so called learning algorithm builds a model of a system under test. The SUT considered is a black box, i.e., we have no information about its internal structure. We do make the assumption that the SUT can be modeled as a deterministic finite state automaton (DFA). We also assume that we know which actions the SUT is able to perform, here called the alphabet Σ . The minimal model of the SUT is the DFA denoted \mathcal{M} . The regular language accepted by the finite state automaton is denoted $\mathcal{L}(\mathcal{M})$, also referred to as U .

The basic set-up for all of the variants of the learning algorithm explained in this section is presented in Figure 19.7. The **Learner** represents the algorithm which is trying to estimate U . The Learner estimates U iteratively through gathering enough pieces of information about U until it is able to construct a hypothesis, also called conjecture or approximation of U . The hypothesis is a DFA \mathcal{A} with the language $\mathcal{L}(\mathcal{A})$. If the conjecture is incorrect the learner will continue to collect information until it can construct a “better” conjecture. The Learner iterates in this fashion until the hypothesis is correct.

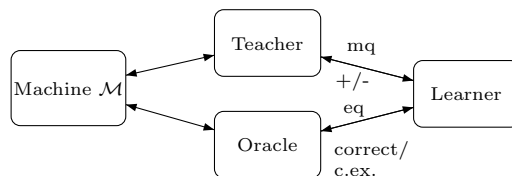


Fig. 19.7. Learning an Automaton.

More specifically, the Learner is able to query the **Teacher** whether a string is accepted by \mathcal{M} or not and the answers will be yes (+) or no (−), seen in Figure 19.7. A query to the Teacher is called **membership query** (mq). The name refers to the question whether a string is a member of $\mathcal{L}(\mathcal{M})$ or not. Furthermore the Learner can ask an **equivalence query** (eq) to an **Oracle** whether the approximation \mathcal{A} is correct or not. If the Oracle deems that the conjecture is equivalent to \mathcal{M} it confirms the correctness of the hypothesis,

otherwise it returns a **counterexample** to \mathcal{A} . The counterexample is in the format of a string which is accepted by \mathcal{M} but not by \mathcal{A} or vice versa.

In the following sections we will describe different algorithms for the Learner, all using the setting described above. The first algorithm, *Observation Packs*, abstracts away the data structure in which a Learner stores the gathered information, using sets instead. In the subsequent section, Section 19.4.2, we present Angluin’s algorithm, in which the observed information is stored in an *Observation Table*. The *Reduced Observation Table* algorithm, presented in Section 19.4.3, is similar to the Observation Table algorithm except it stores less information. Finally, Section 19.4.4 describes the *Discrimination Tree* algorithm which stores information in a binary tree.

19.4.1 Observation Packs

Balcázar et al. abstract from different data structures and present a unified view on the learning problem studied here, storing information in several, called observation packs sets [BDGW97]. An observation pack can be seen as a way of storing pieces of information the Learner has about an unknown regular set.

One piece of information is a so called *observation*. An **observation** is a pair of the form $(s, +)$ or $(s, -)$ for a word s . The **label** $+/-$ signifies the answer to a membership query on string s . The observations in a set must be consistent, i.e., the same word does not appear with different $+/-$ labels.

The observations are organized in finite sets called **components**, which are not necessarily disjoint. A component is denoted C_k , where $k \in \mathbb{N}$ is the component’s index. An **observation pack** \mathcal{O} , or pack for short, is a finite sequence of components, $\mathcal{O} = (C_0, \dots, C_{n-1})$ for some $n \in \mathbb{N}$, for which two conditions must hold:

- OP1 Let $s_k \in C_k$ be the shortest word in C_k ; then s_k is a prefix of all other words in C_k .
- OP2 For each two components C_k and C_l with $l \neq k$, there exists w_{kl} such that both $s_k w_{kl} \in C_k$ and $s_l w_{kl} \in C_l$ but $(C_k, +) \not\iff (C_l, -)$, i.e., they have different labels.

The string s_k is a word that identifies C_k . The set of suffixes for each s_k is defined as $E_k = \{w \mid s_k w \in C_k\}$. So the word w_{kl} mentioned above is in $E_k \cap E_l$. Furthermore $s_k = s_k \varepsilon$ implies $\varepsilon \in E_k$ by the definition of E_k .

We collect in the set S all the shortest words, s_k , from each component and call them **access strings**. The access strings are then used to index both components and sets of suffixes, so the component C_k is C_s and E_k is E_s where $s = s_k$ for some $s \in S$. An observation pack can be identified with the finite set S of access strings and a mapping from S associating to each s the finite set E_s . The set C_s is the set of words sw , for an access string $s \in S$ and suffix $w \in E_s$.

Definition 19.17. A language U agrees with an observation pack \mathcal{O} if all $+$ labels mark words in U , while all $-$ labels mark words not in U .

Assuming that U is a language accepted by a DFA, the suffixes in sets E_s must include evidence (see the second item on the list of conditions on an observation pack) that the access strings belong each to a different equivalence class in the right congruence associated with any regular set U agreeing with the pack. These classes correspond to states of the minimal deterministic finite automaton (DFA) for U : access strings are used to reach the states, hence the name. There can not be any more access strings than states in such an automaton.

Lemma 19.18. *Let \mathcal{O} be a pack, S its set of access strings, U a regular language which agrees with \mathcal{O} , and \mathcal{M} the minimal DFA that recognizes U . Then $|S| \leq |\mathcal{M}|$.*

Proof. Let δ^U and q_0^U be the transition function and the initial state of \mathcal{M} , respectively. Let the mapping f map S into the states of \mathcal{M} in the natural way: from s to $\delta^U(q_0^U, s)$. We prove that this mapping is injective.

Let s and s' be two access strings in the pack and $s \neq s'$. Assume that they both are mapped by f to the same state via the transition function δ^U , i.e. f is not injective. So $\delta^U(q_0^U, s) = q_i$ and $\delta^U(q_0^U, s') = q_i$. Let F^U be the set of accepting states for \mathcal{M} .

According to the properties of the observation pack, there exists a word $w \in E_s \cap E_{s'}$ such that $sw \in U \iff s'w \notin U$. Either, $\delta^U(q_i, w) \in F^U$ or $\delta^U(q_i, w) \notin F^U$. This means

- (1) $\delta^U(q_i, w) \in F^U \implies sw \in U$ and $s'w \in U$, or
- (2) $\delta^U(q_i, w) \notin F^U \implies sw \notin U$ and $s'w \notin U$

But this is a contradiction to $sw \in U \iff s'w \notin U$, so f is injective. \square

Definition 19.19. Let \mathcal{O} be a pack, with access strings S , and U a set agreeing with \mathcal{O} . We say that a word z is *like* $s \in S$ for U if and only if $\forall w \in E_s$ $sw \in U \iff zw \in U$.

To find out whether z is like s we first use the information that sw is labeled $+$ in the pack if $sw \in U$, otherwise $-$. Secondly, for zw we can conduct a membership query and see if $zw \in U$ or not.

Lemma 19.20. *For every word z there is at most one word $s \in S$ such that z is like s for U .*

Proof. Let $s \neq s'$, both in S , and assume z is like s , i.e., $\forall w \in E_s$ $sw \in U \iff zw \in U$. The pack provides a word $w \in E_s \cap E_{s'}$ such that $sw \in U \iff s'w \notin U$. Thus, $zw \in U \iff s'w \notin U$, so that z is not like s' for U . \square

Let $\gamma^{\mathcal{O}, U} : \Sigma^* \rightarrow S$ be the partial function that maps each z to the single access string it is like for U , if there is one; it remains undefined if z is not like any access string for U . From now on, we will use this function in a context in which both \mathcal{O} and U are fixed, so we omit the superscripts and use only γ .

Expanding a Pack Let us now discuss how to extend a pack, evolving to the automaton to learn. We will see the importance of the fact that the function γ can be partial. Let us start by stating that for a given pack we say that a word z is *escaping* when $\gamma(z)$ is undefined.

We can use the knowledge of some escaping string z to adjust the observation pack and get closer to U . In this case we can collect the appropriate observations and expand the pack by adding a new component to it.

The fact that z escapes implies that, for each access string s in the pack, there is a word $sw \in C_s$ providing the suffix w that distinguishes z from s , in the sense that $sw \in U \iff zw \notin U$. A set formed by all such words zw , additionally including z itself, and each labeled by the corresponding $+/-$ label, forms a component that can be added to the pack preserving two mentioned necessary properties OP1 and OP2. Each expansion by one component brings the pack one state closer to the minimal automaton representing U .

Now we want to have a so called *closed pack*, which means that for an access string s and letter a there is no such word sa that escapes. The transition on a from the state represented by s would be undefined if sa escaped. If we discover an escaping word we expand the pack and when there are no more escaping words of this kind we say that the pack is closed.

Definition 19.21. A pack \mathcal{O} agreeing with U is **closed for U** if:

- $\gamma(\varepsilon)$ is defined;
- $\forall s \in S, \forall a \in \Sigma, \gamma(sa)$ is defined.

Note that the definition depends on U since $\gamma = \gamma^{\mathcal{O}, U}$ depends on U . Definition 19.21 actually gives rise to a deterministic finite automaton, whose states are the access strings of the pack, the initial state is $\gamma(\varepsilon)$ and the accepting are states those access strings labeled $+$. We define the transition function to be $\delta(s, a) = \gamma(sa)$ and we extend the transition function δ in a rather common way by, $\delta(s, \varepsilon) = s$, and $\delta(s, wa) = \delta(\delta(s, w), a)$, for $s \in S$, $w \in \Sigma^*$, and $a \in \Sigma$.

Theorem 19.22. *If \mathcal{O} is a pack, U is regular and agrees with \mathcal{O} , and \mathcal{O} has as many components as \mathcal{M} (the minimal DFA that recognizes U) has states, then \mathcal{O} is closed for U so that an automaton can be obtained from \mathcal{O} in the manner above, and furthermore this automaton is isomorphic to \mathcal{M} .*

Proof. By Lemma 19.18, no pack agreeing with U can have more than $|\mathcal{M}|$ components. Therefore, it is not possible to expand \mathcal{O} preserving the agreement with U , and thus it must be closed. Besides, by the cardinality condition, the function f defined and used in Lemma 19.18 becomes bijective. It is routine to show that this function is an isomorphism, that is: 1) it maps $\gamma(\varepsilon)$ to the initial state of \mathcal{M} ; 2) it maps exactly those $s \in S$ with $s \in U$ to the final states; and 3) it commutes with \mathcal{M} 's transition function. \square

Once we are able to form an automaton from our pack, we can make an equivalence query to the Oracle to find out if it is equivalent to \mathcal{M} . If we receive the answer 'yes', we are finished and the automaton is the minimal automaton

that exactly accepts U , otherwise we receive a word that behaves different then the constructed automaton but agrees with \mathcal{M} ; a so called counterexample. Let us study how to process a counterexample.

Counterexample A counterexample is used to correct the hypothesis \mathcal{A} we have about the machine. From the counterexample we will get a word that, when added to the pack, will escape.

Let t be the counterexample of length m , $t = a_0 \dots a_{m-1}$. For $0 \leq i \leq m$ let u_i be the prefix of length i of t and v_i the corresponding suffix, i.e. $t = u_i v_i$.

Let $s_i = \delta(\gamma(\varepsilon), u_i)$ be the state of the automaton based on the pack that is reached by computing on u_i , the initial state be $s_0 = \gamma(\varepsilon)$ and $s_{i+1} = \delta(s_i, a_i)$. The acceptance of t by this automaton is given by whether the final state s_m is accepting, which corresponds to whether $s_m \in U$.

Since the set of strings that are accepted from a state distinguishes a state from another we look upon suffix v_i as an experiment on corresponding state s_i and through membership queries we find out whether $s_i v_i \in U$. The fact that t is a counterexample means that $t \in U \iff s_m \notin U$, where $t = s_0 v_0$ and $s_m = s_m v_m$. So consequently there must exist one or more **breakpoint** positions i such that $s_i v_i \in U \iff s_{i+1} v_{i+1} \notin U$. Since $s_i v_i = s_i a_i v_{i+1}$, the suffix v_{i+1} is an experiment that distinguishes $s_i a_i$ from $s_{i+1} = \delta(s_i, a_i) = \gamma(s_i a_i)$. Now add $s_{i+1} v_{i+1}$ with the appropriate label to the component $C_{s_{i+1}}$. Consequently $\gamma(s_i a_i)$ is not anymore s_{i+1} , hence $s_i a_i$ escapes and we can go on with the pack expansion process.

Example of Observation Pack Let us study the observation pack algorithm applied to the example of the DFA \mathcal{M}_{ex} with the alphabet $\{a, b\}$, shown in Figure 19.8. Initially we conduct an experiment for the empty string. The result of a membership query for ε is $+$ and the first component is initialized with this observation, $C_0 = \{(\varepsilon, +)\}$. The component's corresponding suffix-set is $E_0 = \{\varepsilon\}$. The evolution of \mathcal{O} for this example can be viewed in Table 19.1. The sign $-$ in the table means that the component is unchanged.

In the next step we want to close the pack, therefore we ask membership queries for εa and εb . The results are the observations $(a, +)$ and $(b, +)$, whose strings are like ε , the access string for component C_0 . The pack is now closed and we can construct a corresponding automaton \mathcal{A}^0 , which can be seen in Figure 19.9.

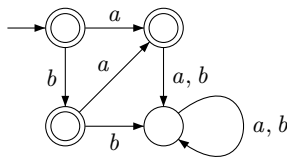


Fig. 19.8. The machine \mathcal{M}_{ex}

We conduct an equivalence query for \mathcal{A}^0 in order to see if the conjecture is correct. As answer we get a counterexample $t = ab$, we see that $ab \notin \mathcal{L}(\mathcal{M}_{ex})$ but $ab \in \mathcal{L}(\mathcal{A}^0)$. The counterexample can be divided into prefix and suffix, $u_0 = \varepsilon$ and $v_0 = ab$ in order to find the breakpoint where \mathcal{M}_{ex} and \mathcal{A}^0 behave differently. A state in a hypothesis is called s_i , for this counterexample $0 \leq i \leq 2$. We see that a breakpoint can be found for $i = 0$ since $s_0 ab \notin \mathcal{L}(\mathcal{M}_{ex})$ but $s_1 b \in \mathcal{L}(\mathcal{M}_{ex})$ (Recall that $s_1 = \delta(s_0, a)$). Thus b is the experiment that distinguishes $s_0 a = \gamma(\varepsilon)a = a$ from $s_1 = \gamma(\varepsilon a) = \varepsilon$.

	Step 0	Step 1
C_0, E_0	$C_0 = \{(\varepsilon, +)\}, E_0 = \{\varepsilon\}$	$C_0 = \{(\varepsilon, +), (b, +)\}, E_0 = \{\varepsilon, b\}$
C_1, E_1		$C_1 = \{(a, +), (ab, -)\}, E_1 = \{\varepsilon, b\}$
C_2, E_2		
C_3, E_3		
	Step 2	Step 3
C_0, E_0	–	–
C_1, E_1	–	$C_1 = \{(a, +), (ab, -), (aa, -)\},$ $E_1 = \{\varepsilon, b, a\}$
C_2, E_2	$C_2 = \{(aa, -)\}, E_2 = \{\varepsilon\}$	–
C_3, E_3		$C_3 = \{(b, +), (ba, +), (bb, -)\},$ $E_3 = \{\varepsilon, a, b\}$

Table 19.1. The observation pack

It is now enough to add $(b, +)$ to the component C_ε , so $C_0 = \{(\varepsilon, +), (b, +)\}$ and $E_0 = \{\varepsilon, b\}$. Now $\gamma(a)$ is no longer s_0 since a escapes and therefore we expand the pack with a new component C_1 , where $C_1 = \{(a, +), (ab, -)\}$ and $E_1 = \{\varepsilon, b\}$, see Step 1. The mapping of b to an access string is now changed to $\gamma(b) = a$.

The next step is to make the pack closed, the missing words are aa and ab . Using membership queries we try to discover an existing access string aa behaves like, but we cannot find one, so it escapes. From the observation $(aa, -)$ we create a new component, $C_2 = \{(aa, -)\}$, whose corresponding suffix set is $E_2 = \{\varepsilon\}$, see Step 2. (The suffix ε differentiates s_2 from all other access strings, so no further suffixes need to be added to C_2 .) The next word to map into an access string is ab and we see that $\gamma(ab) = aa$.

Since we now created a new component C_2 we must make sure that the pack is closed, hence we have to check to what access strings the strings aaa and aab are mapped. Checking these yields $\gamma(aaa) = aa$ and $\gamma(aab) = aa$. The observation pack is now closed and it is possible to form a hypothesis, \mathcal{A}^1 , about the machine, see Figure 19.9.

Now we conduct an equivalence query for the hypothesis \mathcal{A}^1 . The Oracle returns a counterexample $t = ba$. Again we perform the search for a breakpoint in t , we initialize the prefix and suffix of t to be $u_0 = \varepsilon$ and $v_0 = ba$, respectively. The breakpoint is found for $i = 0$ where $s_0 ba \in \mathcal{L}(\mathcal{M}_{ex})$ but $s_1 a \notin \mathcal{L}(\mathcal{M}_{ex})$.

In order to adjust \mathcal{A}^1 , we add $(aa, -)$ to component $(C_{s_1} = C_a)C_1$, transforming it into $C_1 = \{(a, +), (ab, -), (aa, -)\}$. Now $\gamma(b)$ is not anymore a (it does not behave as a on suffix a) but is instead undefined. This implies that we

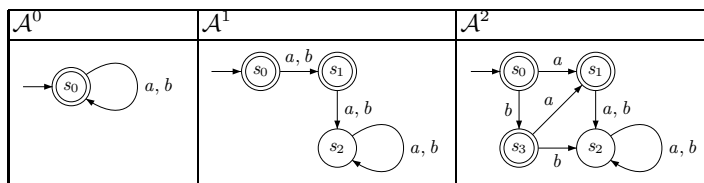


Fig. 19.9. The machine's approximations

have to create a new component, C_3 , based on the escaping string b , and add it to \mathcal{O} . In order to distinguish the access strings s_0 and s_1 , we add to E_3 , the suffix a to distinguish b from s_1 , and the suffix b to distinguish b from s_0 . (The string ε distinguishes b from s_2 .) The new component is $C_3 = \{(b, +), (ba, +), (bb, -)\}$ with corresponding suffix set $E_3 = \{\varepsilon, a, b\}$, see Step 3.

We are now able to create our next hypothesis \mathcal{A}^2 and conduct an equivalence query for it. The answer to this query is 'yes' and the algorithm terminates and outputs the correct automaton \mathcal{A}^2 , seen in Figure 19.9.

19.4.2 Angluin's Algorithm

The learning algorithm by Balcázar et al is rather abstract, putting the selected information in sets. In this section we discuss a learning algorithm that still puts information into sets but uses a more concrete data structure: a table. The observation pack algorithm does not tell us how to store exactly all information we query for. In the algorithm we will introduce next, we see how it is possible to store all information in an easy manner. We will now present the *Angluin Algorithm*, which we will also refer to as the *Observation Table Algorithm* [Ang87].

The algorithm, or Learner, makes use of the same environment and plays the same roll as described in Section 19.4. Initially the algorithm has no knowledge of the SUT's regular language, but the information it accumulates about the behavior of the machine, is entered into a so called *observation table*. Due to the table the algorithm has at any point in time information about a finite collection of strings over a known finite alphabet Σ , classifying them as members or non-members of some unknown regular set U .

Angluin's algorithm will make sure that the observation table fulfills some criteria before it constructs a deterministic finite-state machine \mathcal{A} from information in the observation table. This hypothesis of what the language of the SUT is, is the Learner's conjecture which will be sent to the Oracle.

The Observation Table The information accumulated by the algorithm is a finite collection of observations, which is organized into an observation table. The table is defined as follows:

Definition 19.23. An **Observation Table** over a given alphabet Σ is a tuple $\mathcal{OT} = (S_A, E_A, T_A)^1$, where

¹ The index A signifies that the sets belong to Angluin's algorithm.

- $S_A \subseteq \Sigma^*$ is a nonempty finite prefix-closed set,
- $E_A \subseteq \Sigma^*$ is a nonempty finite suffix-closed set, and
- $T_A : ((S_A \cup S_A \cdot \Sigma) \times E_A) \rightarrow \{+, -\}$ is a (finite) function

satisfying the property that $se = s'e'$ implies $T_A(s, e) = T_A(s', e')$ for $s, s' \in S_A \cup S_A \cdot \Sigma$ and for all $e, e' \in E_A$.

The words in $S_A \cup S_A \cdot \Sigma$ are called *row labels* and the words in E_A are called *column labels*. The entries consists of signs (+/-) representing whether a word is accepting or not.

The observation table is divided into an upper part and a lower part. The upper part of the observation table is indexed by row labels in S_A . They play a role similar to access strings in the observation pack algorithm, see Section 19.4.1. The lower part's row labels are indexed by all strings of the form sa , $a \in \Sigma$ and $s \in S_A$, unless they already appear in the upper part. Moreover the table is indexed column wise by a suffix-closed set E_A of strings. The suffixes are used in the same fashion as in Section 19.4.1, to distinguish a access string/row from another. The function T_A maps a row label s and a column label e , i.e. $T_A(s, e)$, to the set $\{+/-\}$, it is defined to be + if $se \in U$ and - otherwise. (Note that T_A is total.)

A function $row(s)$ for every $s \in (S_A \cup S_A \cdot \Sigma)$ denotes the finite function $f : E_A \rightarrow \{+, -\}$ defined by $f(e) = T_A(s, e)$. In other words, $row(s)$ is the row of entries in the observation table for index s .

It is possible that there exists an entry on a string in several places in the table due to the fact that a string can be divided into different suffixes and prefixes, i.e. row and column labels. Of course, these labels have to agree. This is required by Definition 19.23.

A distinct row in \mathcal{OT} characterizes a state in the automaton which can be constructed from \mathcal{OT} . All the row labels to unique rows must be kept in S_A . The rows labeled by elements of $S_A \cdot \Sigma$ are used to create the transition function for the automaton.

To construct a DFA from the observation table it must fulfill two criteria. It has to be *closed* and *consistent*.

Definition 19.24. An observation table \mathcal{OT} is **closed** if for each $s \in S_A \cdot \Sigma$ there exists an $s' \in S_A$ such that $row(s) = row(s')$.

Definition 19.25. An observation table is **consistent** if whenever $row(s) = row(s')$ for $s, s' \in S_A$ then $row(sa) = row(s'a)$ for all $a \in \Sigma$.

When the observation table (S_A, E_A, T_A) is closed and consistent it is possible to construct the corresponding DFA $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ as follows:

- $Q = \{row(s) \mid s \in S_A\}$,
- $q_0 = row(\varepsilon)$,
- $F = \{row(s) \mid s \in S_A \text{ and } T_A(s, \varepsilon) = +\}$,
- $\delta(row(s), a) = row(sa)$.

The corresponding DFA constructed in this manner from table (S_A, E_A, T_A) is denoted $\mathcal{A}(S_A, E_A, T_A)$. The first property, closed, holds if a row representing a successor state of some state in Q , the successor state is also in Q . The second property, consistent, ensures that if two rows represent the same state, then they must also have the same successor state on all input symbols.

The Learning Algorithm The learning algorithm, Algorithm 1, maintains the observation table \mathcal{OT} . The sets S_A and E_A are both initialized to $\{\varepsilon\}$. Next the the algorithm performs membership queries for ε and for each $a \in \Sigma$, the result is a label for each queried string. The observation table \mathcal{OT} is initialized to (S_A, E_A, T_A) .

Algorithm 1 Angluin's Learning Algorithm.

```

1 Function Angluin()
2 begin
3   Initialize  $S_A$  and  $E_A$  to  $\{\varepsilon\}$ .
4   Ask membership queries for  $\varepsilon$  and each  $a \in \Sigma$ .
5   Construct the initial observation table  $(S_A, E_A, T_A)$ .
6
7 repeat:
8   while  $(S_A, E_A, T_A)$  is not closed or not consistent:
9     if  $(S_A, E_A, T_A)$  is not consistent,
10      then find  $s$  and  $s'$  in  $S_A$ ,  $a \in \Sigma$ , and  $e \in E_A$  such that
11         $row(s) = row(s')$  and  $T_A(sa, e) \neq T_A(s'a, e)$ ,
12        add  $ae$  to  $E_A$ ,
13        and extend  $T_A$  to  $(S_A \cup S_A \cdot \Sigma) \cdot E_A$  using membership queries.
14
15      if  $(S_A, E_A, T_A)$  is not closed,
16        then find  $s \in S_A$  and  $a \in \Sigma$  such that
17           $row(sa)$  is different from  $row(s')$  for all  $s' \in S_A$ ,
18          add  $sa$  to  $S_A$ ,
19          and extend  $T_A$  to  $(S_A \cup S_A \cdot \Sigma) \cdot E_A$  using membership queries.
20
21      Once  $(S_A, E_A, T_A)$  is closed and consistent, let  $\mathcal{A} = \mathcal{A}(S_A, E_A, T_A)$ .
22      Make an equivalence query to the Oracle with the hypothesis  $\mathcal{A}$ .
23      if the Oracle replies with a counterexample  $t$ ,
24        then add  $t$  and all its prefixes to  $S_A$ 
25        and extend  $T_A$  to  $(S_A \cup S_A \cdot \Sigma) \cdot E_A$  using membership queries.
26 until the Oracle replies 'yes' to the hypothesis  $\mathcal{A}$ .
27 return  $\mathcal{A}$ .
28 end

```

Next the algorithm makes sure that \mathcal{OT} is closed and consistent. If \mathcal{OT} is not consistent, one inconsistency is resolved through finding two strings $s, s' \in S_A$, $a \in \Sigma$ and $e \in E_A$ such that $row(s) = row(s')$ and $T_A(s, ae) \neq T_A(s', ae)$ and adding this new suffix ae to E_A . The observation table is consistent when no more strings as these can be found. The algorithm fills the missing entries in the new column by asking membership queries.

If \mathcal{OT} is not closed the algorithm finds $s \in S_A$ and $a \in \Sigma$ such that $row(sa) \neq row(s')$ for all $s' \in S_A$. The algorithm makes the table closed by adding sa to S_A . When no more such strings can be found the table is closed. The missing entries in \mathcal{OT} are inserted through membership queries.

When \mathcal{OT} is closed and consistent the hypothesis $\mathcal{A} = \mathcal{A}(S_A, E_A, T_A)$ can be formed and its correctness checked through an equivalence query to the Oracle. The Oracle can either reply with a counterexample t , such that $t \in \mathcal{L}(\mathcal{M}) \iff t \notin \mathcal{L}(\mathcal{A})$, or 'yes'. If the answer is 'yes' the algorithm halts and outputs the correct conjecture \mathcal{A} . Otherwise t is a counterexample. In contrast to finding a breakpoint as in the observation pack algorithm, Angluin's approach is to add t and all its prefixes to the table. Then all missing entries are filled. In this way, also the prefix that would be identified by finding the breakpoint is processed.

Example of Observation table The machine \mathcal{M}_{ex} we want to learn using Algorithm 1, is shown in Figure 19.8. The algorithm initializes \mathcal{OT} (lines 3–5) to A^0 in Table 19.2. Table A^0 is closed and consistent (line 8). Therefore the algorithm can form an automaton based on it (line 21), resulting in \mathcal{A}^0 , Figure 19.10.

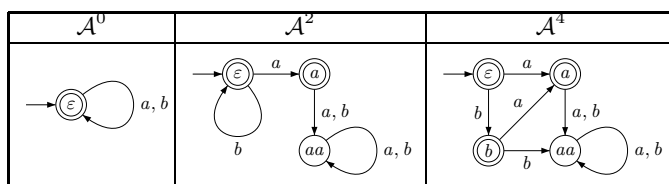


Fig. 19.10. The machine's approximations

The next step is to do an equivalence query for \mathcal{A}^0 (line 22). The answer from the Oracle is the counterexample aa since $aa \notin \mathcal{L}(\mathcal{M}_{ex})$ but $aa \in \mathcal{L}(\mathcal{A}^0)$. The counterexample and its prefixes are added to S_A , in the table representation the upper part of the table, and the lower part of the table is extended. The strings in S_A are a and aa and in $S_A \cdot \Sigma$ are b, ab, aaa and aab , see A^1 in Table 19.2. The membership queries for the new entries are made and the answers inserted.

Step 0	Step 1	Step 2	Step 3	Step 4
A^0	A^1	A^2	A^3	A^4
ε	ε	ε	ε	ε
ε	ε	ε	ε	ε
a	a	a	a	a
b	aa	aa	b	b
	b	b	aa	aa
	ab	ab	bb	bb
	aaa	aaa	ab	ab
	aab	aab	ba	ba
			aaa	aaa
			aab	aab
			bba	bba
			bbb	bbb

Table 19.2. The observation tables

The algorithm returns (line 8) to check again that the observation table is closed and consistent. This time it will discover an inconsistency in A^1 due to $row(\varepsilon a) \neq row(aa)$, the lefthand side being $+$ and the right hand side being $-$. A new suffix a , which distinguishes the two inconsistent rows a and ε , are added to E_A (line 12). The empty entries in the new columns are filled and the result is A^2 in Table 19.2.

Table A^2 is next checked that it is closed (line 15). Since no row label in the lower part of the table does not already exist in the upper part it is closed. It is now possible to form the automaton \mathcal{A}^2 , showed in Figure 19.10.

Next the algorithm performs an equivalence query to the Oracle with the hypothesis \mathcal{A}^2 (line 22). The response given is again a counterexample, this time $t = bb$, since $bb \notin \mathcal{L}(\mathcal{M}_{ex})$ but $bb \in \mathcal{L}(\mathcal{A}^2)$. The string bb and its prefixes are added to S_A . The lower part of the table is extended by adding the new row labels ba , bba and bbb . The algorithm fills all the empty entries by executing membership queries. This yields table A^3 in Table 19.2.

In the last step the algorithm finds one more inconsistency, due to $row(\varepsilon b) \neq row(bb)$. Solving the inconsistency yields the new column label b , which is added to E_A . The resulting table A^4 , see Table 19.2, is closed and consistent and the corresponding hypothesis, \mathcal{A}^4 in Figure 19.10, returns a 'yes' in the final equivalence check. The algorithm returns \mathcal{A}^4 and halts.

19.4.3 Reduced Observation Tables

We have so far seen two proposals of learning algorithms, the observation pack and the observation table (or Angluin's) algorithms. The next algorithm we will present is a Learner closely related to Angluin's algorithm.

In the setting of the observation table algorithm, see Section 19.4.2, the observation table is likely to contain several rows representing one state. The algorithm presented here is based on the observation table algorithm but contains a smaller version of the table. We will refer to this algorithm by the name *Reduced Observation Table Algorithm*, introduced by Rivest and Schapire [RS93].

Many notions of Angluin's algorithm can directly be transferred to the reduced observation table algorithm. In view of how a table is constructed the sets S_A , E_A , and the table function T_A correspond directly to S_R , E_R , and T_R , respectively. The entries, row labels, column labels and rows are also to be interpreted as in Angluin's algorithm. As in the case of the observation table algorithm, the information the Learner accumulates is a finite collection of observations, which is organized into a *reduced* observation table, denoted \mathcal{ROT} . The table is defined as follows:

Definition 19.26. A tuple $\mathcal{ROT} = (S_R, E_R, T_R)$ over a given alphabet Σ is a **Reduced Observation Table**, where

- $S_R, E_R \subseteq \Sigma^*$ are nonempty finite sets,
- $T_R : ((S_R \cup S_R \cdot \Sigma) \times E_R) \rightarrow \{+, -\}$ is a (finite) function,
- $se = s'e'$ implies $T_R(s, e) = T_R(s', e')$ for $s, s' \in S_R \cup S_R \cdot \Sigma$ and for all $e, e' \in E_R$, and

- $row(s) = row(s')$ implies $s = s'$ for all $s, s' \in S_R$.

Thus, a reduced observation table differs from an observation table in two ways. First, S_R does not need to be prefix-closed. Second, every row appears only once in the upper part of the table.

Since there cannot be two row labels in S_R that map to the same row, there is no need to check a reduced observation table for consistency. In other words, there cannot be any inconsistency.

The reduced observation table algorithm contains only the access strings, recall Section 19.4.1, which results in a smaller table than Angluin's. This in turn is the cause for using less membership queries. The definition of a **closed** \mathcal{ROT} is as for the observation table. From a closed \mathcal{ROT} we can construct an automaton in the same manner as in Angluin's algorithm. Besides containing a smaller observation table, a second source of efficiency, compared to Angluin's algorithm, is the faster processing of counterexamples.

Processing a counterexample $t = u_i v_i$ of length m , recall Section 19.4.1, means finding a breakpoint i such that $s_i v_i \in U \iff s_{i+1} v_{i+1} \notin U$, where the $s_i = \delta(row(\varepsilon), u_i)$ are the states visited by t along the automaton and v_i are the corresponding suffixes of t . Some such breakpoint must exist since $s_0 v_0 \in U \iff s_m v_m \notin U$, so that a sequential search will find, say, the first one with m membership queries. Rivest and Schapire show how a binary search finds a breakpoint with $\log m$ queries.

The reduced table is kept small by not adding all prefixes of the counterexample as rows. This means that the new automaton may still classify the previous counterexample incorrectly, so the same counterexample can potentially be used to answer several equivalence queries. Two equal counterexamples can also occur in an algorithm which uses so called discrimination trees, to be discussed in Section 19.4.4.

The Learning Algorithm The reduced observation table algorithm is basically constructed in the same manner as the observation table algorithm. The difference is that since only unique rows are contained in the upper part of the table of the reduced observation table algorithm there is no need to check for consistency. The handling of a counterexample is different to Angluin's algorithm. In this algorithm we search for a breakpoint in the counterexample and add only one row to the upper part of the table, not every prefix of the counterexample as in the case of the other algorithm. The Reduced Observation Table algorithm is given in Algorithm 2.

Example of Reduced Observation Table We will here present an example how the reduced observation table evolves when learning the same example as in Section 19.4.2 shown in Figure 19.8. The table is initialized in the same manner as in Angluin's algorithm so S_R and E_R is set to ε (line 3). For all actions in Σ and ε we perform membership queries and add them to the lower part of the table (lines 4-5), see result in A^0 in Table 19.3. Since the table always is

Algorithm 2 Reduced Observation Table Learning Algorithm.

```

1 Function Reduced – Observation – Table()
2 begin
3   Initialize  $S_R$  and  $E_R$  to  $\{\varepsilon\}$ .
4   Ask membership queries for  $\varepsilon$  and each  $a \in \Sigma$ .
5   Construct the initial reduced observation table  $(S_R, E_R, T_R)$ .
6
7 repeat:
8   while  $(S_R, E_R, T_R)$  is not closed:
9     then find  $s \in S_R$  and  $a \in \Sigma$  such that
10       $row(sa)$  is different from  $row(s')$  for all  $s' \in S_R$ ,
11      add  $sa$  to  $S_R$ ,
12      and extend  $T_R$  to  $(S_R \cup S_R \cdot \Sigma) \cdot E_R$  using membership queries.
13
14   Once  $(S_R, E_R, T_R)$  is closed, let  $\mathcal{A} = \mathcal{A}(S_R, E_R, T_R)$ .
15   Make an equivalence query to the Oracle with the hypothesis  $\mathcal{A}$ .
16   if the Oracle replies with a counterexample  $t$ 
17     then let the counterexample  $t$  be  $u_i v_i$ , where  $u_0 = v_m = \varepsilon$  and  $v_0 = u_m = t$ 
18     and  $t = u_i a_i v_{i+1}$  for  $i < m$ , and  $m$  is the length of  $t$ .
19     Find a breakpoint position  $i$  for which  $s_i a_i v_{i+1} \in U \iff s_{i+1} v_{i+1} \notin U$  holds,
20     where  $s_i$  are the states visited by  $t$  along  $\mathcal{A}$ .
21     Add  $v_{i+1}$  to  $E_R$ 
22     and extend  $T_R$  to  $(S_R \cup S_R \cdot \Sigma) \cdot E_R$  using membership queries.
23   until the Oracle replies 'yes' to the hypothesis  $\mathcal{A}$ .
24   return  $\mathcal{A}$ .
25 end

```

consistent we only have to check that the reduced observation table is closed (line 8). There is no row in the lower part of the table which is not already contained in the upper part, hence the table is closed. It is now possible to form an automaton from the information in the table (line 14), the result is shown in \mathcal{A}^0 , Figure 19.11.

Step 0	Step 1	Step 2	Step 3	Step 4
A^0	A^1	A^2	A^3	A^4
ε	ε b	ε b	ε b	ε b a
ε +	ε + +	ε + +	ε + +	ε + + +
a +	a + -	a + -	a + -	a + - -
b +	b + -	b + -	aa - -	b + - +
		aa - -	b + -	aa - - -
		ab - -	ab - -	ab - - -
			aaa - -	ba + - -
			aab - -	bb - - -
				aaa - - -
				aab - - -

Table 19.3. The reduced observation tables

The Learner will thereafter make an equivalence query to the Oracle, which returns a counterexample $t = ab$, which we divide into prefix $u_0 = \varepsilon$ and suffix $v_0 = ab$ (lines 17–18). We search for the breakpoint in the counterexample and find it for $i = 0$ since $s_0 ab \notin \mathcal{L}(\mathcal{M}_{ex}) \iff s_1 b \in \mathcal{L}(\mathcal{M}_{ex})$. Now we can add the new column label b (line 21). The result of this operation is shown as A^1 in Table 19.3, now $row(\varepsilon) \neq row(a)$.

Next we check whether table A^1 is closed. The Learner discovers that there is one row in the lower part of the table which is not in the upper part (lines 9–10). In order to rectify this we add the row of a to the upper part and in the lower part we add its successor states, shown in Table 19.3, Table A^2 . The extension to the table gives rise to a non-closed table again. So for this reason we move row of aa in the same manner to the upper part, see the result in Table A^3 . Now the table is closed and automaton \mathcal{A}^3 can be formed, shown in Figure 19.11. The Learner queries the Oracle with this hypothesis, but the Oracle answers with a counterexample.

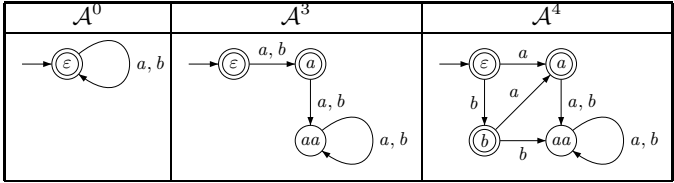


Fig. 19.11. The machine’s approximations

The counterexample is $t = ba$, where $ba \in \mathcal{L}(\mathcal{M}_{ex})$ but $ba \notin \mathcal{L}(\mathcal{A}^3)$. Finding the breakpoint is the process of following \mathcal{A}^3 along t and discovering where the mistake of not accepting t is made. The breakpoint in this case is found for $i = 0$ since $s_0ba \in \mathcal{L}(\mathcal{M}_{ex})$ but $s_1a \notin \mathcal{L}(\mathcal{M}_{ex})$ where the states s_0 and s_1 are represented by ϵ and a , respectively. We add the new column label a to \mathcal{ROT} and extend the lower part of the table.

In the loop of checking if \mathcal{ROT} is closed, the Learner will discover that the row of b does not exist in the upper part of the table. This row is moved to the upper part of the table and the lower part is extended. The result shown in A^4 , Table 19.3. This table is closed and with a final equivalence check with the corresponding automaton \mathcal{A}_4 in Figure 19.11 as hypothesis the answer from the Oracle is a ‘yes’ and the algorithm terminates.

19.4.4 Discrimination Trees

In this section we will discuss a fourth approach to implementing the Learner in the setting described earlier. Instead of using sets for storing the Learner’s observations, as in the observation pack algorithm, we will show how it is possible to use a tree.

The Learner’s data structure is in this case a binary tree with labeled nodes.

Definition 19.27. Given an alphabet Σ , a **discrimination tree** is tuple $\mathcal{DT} = (S_D, E_D, t)$ where

- $S_D, E_D \subseteq \Sigma^*$ are nonempty finite sets of access strings and suffixes, respectively,

- t is an $S_D \cup E_D$ -labeled binary tree where,
 - non-leaf nodes are labeled with suffixes in E_D , and
 - leaves are labeled with access strings in S_D .

The information about whether a string is accepted or not is contained in the structure of a discrimination tree. The computation of the function $\gamma(w)$ for a string w - recall that $\gamma(w)$ maps w to the only access string w is like - becomes simple with the use of discrimination trees; traverse the tree on w in the following manner. Query wv for membership in each node labeled v and enter the right child node on a positive answer and left otherwise. The calculation of $\gamma(w)$ stops in a leaf which is labeled with an access string. The function γ is total since the computation can be done for any input string w . Let *Sift* be the function that traverses a given discrimination tree for input string w in the just described manner, starting at the root.

A discrimination tree is initialized with the root labeled by ε and two leaves, one labeled by ε and the other one with a string which answers the opposite to ε on a membership query.

The discrimination tree is always closed and consistent, therefore escaping strings can only be obtained through counterexamples. Given a discrimination tree \mathcal{DT} , the escaping string is taken via the shortest prefix u_i for which $s_i a_i v_{i+1} \in U \iff s_{i+1} v_{i+1} \notin U$ holds, where $s_i = \text{Sift}(u_i, \mathcal{DT})$, see Section 19.4.1 on how to handle counterexamples. The leaf s_{i+1} is replaced by an internal node labeled v_{i+1} which will separate the old leaf representing state s_{i+1} from the new one $s_i a_i$. The algorithm can now start its next iteration with another equivalence query. Altogether, note that equivalence queries are used to modify the tree in order to derive a hypothesis of the automaton to learn, while membership queries are used in the translation from the tree to an automaton and for processing a counterexample.

The Learning Algorithm The discrimination tree algorithm consists of the main function *Discrimination-Tree* and auxiliary helper functions; *Sift*, *Hypothesis*, and *Update-Tree*. The complete algorithm is shown in Algorithm 3, [KV94, BDGW97].

The main function *Discrimination-Tree* (line 1) first initializes the discrimination tree. It performs a membership query for ε and if the answer to this query is positive, meaning $\varepsilon \in U$, then the first hypothesis \mathcal{A} has one accepting state, otherwise one non-accepting. This state is the initial state. All the actions from this state will loop back to this state.

The next step for the Learner is to execute an equivalence query with this conjecture. With the information about the counterexample, which the Learner receives, the discrimination tree will be initialized, the root labeled with ε and the leaves labeled with ε and the counterexample, t , from the Oracle.

Henceforth the main function will enter a loop; construct a hypothesis from the discrimination tree and conduct an equivalence query on it. The function processes a counterexample or stops if the Oracle accepts the hypothesis.

The helper function *Sift* (line 31) returns an access string for a given string w by simply sifting down the given tree \mathcal{DT} on w , as described earlier.

The helper function *Hypothesis* (line 44) constructs the hypothesis given a discrimination tree. The hypothesis \mathcal{A} has for each leaf a state, the states are labeled with the access strings, and ε is the initial state. The transitions are constructed in the following manner. For each state s and each $a \in \Sigma$ sift down the discrimination tree on sa , direct the outgoing edge labeled a to the result of the sift action.

The last helper function *Update-Tree* (line 56), updates the discrimination tree given a counterexample t and a discrimination tree \mathcal{DT} . The function finds the breakpoint in the counterexample and replaces the erroring leaf by a new node. The replaced leaf becomes one leaf to the new node and the other leaf is a suffix of t .

Example of Discrimination Trees We will here present an example of how the discrimination tree algorithm works. The example machine we will learn is shown in Figure 19.8. The alphabet for the machine is as mentioned earlier $\{a, b\}$.

The first step in the algorithm is to do a membership query for ε to determine whether it is accepting or not, see function *Discrimination-Tree* in Algorithm 3 (line 3). In the succeeding step we construct the automaton \mathcal{A}^0 , shown in Figure 19.12, with one state where the transitions of a and b loop back to the initial state (line 4). The state is accepting since ε is accepting. Now we can conduct an equivalence query for the first hypothesis \mathcal{A}^0 .

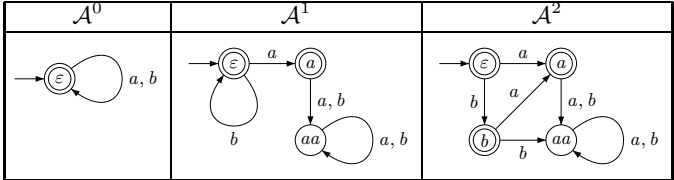


Fig. 19.12. The machine’s approximations

An equivalence check for \mathcal{A}^0 yields a counterexample $t = aa$. We now have the information we need in order to initialize the discrimination tree (line 8). The root is set to be labeled with the distinguishing string ε and two leaves with ε and the counterexample aa , see tree A^0 in Figure 19.13. After we update the tree with the counterexample we get the discrimination tree A^1 shown in Figure 19.13.

We now construct the automaton corresponding to A^1 . It is created by letting every leaf in the tree be a state in the automaton. Given a discrimination tree \mathcal{DT} , the transition for an action a in state s is $\delta(s, a) = (\gamma(sa) =)Sift(sa, \mathcal{DT})$. The tree A^0 has the corresponding automaton \mathcal{A}^1 in Figure 19.12.

Algorithm 3 Discrimination Tree Learning Algorithm.

```

1  Function Discrimination - Tree()
2  begin
3  Ask a membership query for  $\varepsilon$ .
4  Construct hypothesis  $\mathcal{A}$  with one state and self-loops for all  $a \in \Sigma$ .
5  If  $\varepsilon$  is accepted the state is accepting, otherwise not.
6  Make an equivalence query with  $\mathcal{A}$ ; If unsuccessful let the counterexample be  $t$ .
7
8  Initialize the tree  $\mathcal{DT}$  to have the root labeled with  $\varepsilon$  and the leaves labeled with  $\varepsilon$  and  $t$ .
9  Update - Tree( $t, \mathcal{DT}$ ).
10 repeat:
11   Let  $\mathcal{DT}$  be the current discrimination tree and,
12   let  $\mathcal{A} = \text{Hypothesis}(\mathcal{DT})$ .
13   Make the equivalence query with  $\mathcal{A}$ .
14   if yes,
15     then halt and output  $\mathcal{A}$ .
16   else
17     let  $t$  be the counterexample.
18     Update - Tree( $t, \mathcal{DT}$ ).
19 end

31 Function Sift( $w, \mathcal{DT}$ )
32 begin
33 Set the current node to be the root node of  $\mathcal{DT}$ .
34 repeat:
35   Let  $v$  be the distinguishing string at the current node in the tree.
36   Make a membership query for  $wv$ .
37   if  $wv$  is accepted,
38     then update current node to be the right child of the current node.
39   else
40     update current node to be the left child of the current node.
41   if current node is a leaf node,
42     then return the access string stored at this leaf.
43 end

44 Function Hypothesis( $\mathcal{DT}$ )
45 begin
46 for each leaf (access string) of  $\mathcal{DT}$ ,
47   create a state in  $\mathcal{A}$  that is labeled by that leaf (access string).
48 Let the initial state be  $\varepsilon$ .
49
50 for each access string  $s$  of  $\mathcal{A}$  and each  $a \in \Sigma$ ,
51 compute the  $a$ -transition from state  $s$  as follows:
52   Let  $s' = \text{Sift}(sa, \mathcal{DT})$  and,
53   let  $\delta(s, a) = s'$ .
54 return  $\mathcal{A}$ .
55 end

56 Function Update - Tree( $t, \mathcal{DT}$ )
57 begin
58 Let the counterexample  $t$  be  $u_i v_i$ , and  $t = u_i a_i v_{i+1}$  for  $i < m$ , where  $m$  is the length of  $t$ .
59 Find the shortest prefix  $u_i$  for which  $s_i a_i v_{i+1} \in U \iff s_{i+1} v_{i+1} \notin U$  holds,
60 where  $s_i = \text{Sift}(u_i, \mathcal{DT})$  and  $s_{i+1} = \text{Sift}(u_{i+1}, \mathcal{DT})$ .
61
62 Replace the leaf  $s_{i+1}$  by an internal node labeled  $v_{i+1}$ ,
63 let one of the leaves be the replaced leaf's label and let the other be  $s_i a_i$ .
64 end

```

In the next step we make an equivalence query for \mathcal{A}^1 to investigate if the hypothesis is correct. We receive a counter example $t = bb$ since $bb \notin \mathcal{L}(\mathcal{M}_{ex})$ but $bb \in \mathcal{L}(\mathcal{A}^1)$. The example is divided into prefix and suffix where $u_0 = \varepsilon$ and $v_0 = bb$. The breakpoint is found for $i = 0$, where $s_0 bb \notin \mathcal{L}(\mathcal{M}_{ex})$ but $s_1 b \in \mathcal{L}(\mathcal{M}_{ex})$. Now we will update the tree in order for it to act correct in relation to the counterexample.

As described in function *Update-Tree* we sift down the tree on b and stop in the leaf ε . We replace this leaf by a node labeled b and let the leaves of b be the replaced leaf ε and b , see resulting tree \mathcal{A}^2 in Figure 19.13. The corresponding automaton for the updated tree is \mathcal{A}^2 in Figure 19.12.

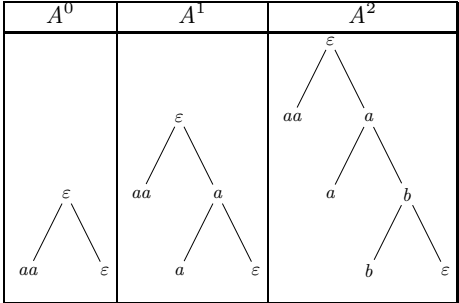


Fig. 19.13. The discrimination trees

Finally, we conduct a equivalence query for hypothesis \mathcal{A}^2 . We receive the answer 'yes' from the Oracle and the algorithm terminates.

19.4.5 Equivalence Check

The Oracle resolves an equivalence check in the learning setting that we discuss. To learn automata in practice, a realization of such an oracle has to be provided.

The VC-algorithm Vasilevskii and Chow presented independently a method for comparing the language of two automata where one is given as a black-box, provided an upper bound on the number of states is given [Vas73, Cho78].

Of course, two languages L_1 and L_2 are equal, iff they contain the same strings. Comparing an infinite number of strings, however, yields no effective algorithm.

For a regular language, we know that if the length of a string exceeds the number of states of the automaton defining the language, at least one state must be visited twice. In other words, the language of a finite state machine can be described by a finite set of strings together with some "pumping" information.

Using this observation, one can show that it suffices to compare L_1 and L_2 for all strings up to some length linearly bounded by the sizes of the two automata.

Vasilevskii and Chow further show that if one automaton is given explicitly, the number of comparisons can be slightly improved.

A probabilistic approach Angluin [Ang87] proposes an equivalence check that yields a correct answer up to some given failure probability.

An Oracle can be realized as a function picking strings randomly and comparing the machine and the hypothesis for those. If a mismatch is found, the corresponding string is a counterexample. If no mismatch is found, the two systems are classified as identical. This conclusion might be wrong with a certain probability. However, if we know the probability distribution of strings being accepted, one can compute the number of comparisons needed to guarantee that this failure probability is below a given limit. See [Ang87] for details.

19.4.6 Query Complexity of the Algorithms

We discuss only the query complexity of the algorithms, i.e., the number of queries needed to construct a correct model of the SUT's regular language. Their time complexity can be estimated with similar arguments.

In this subsection, let n , m , and k be the number of states of \mathcal{M} , the length of the longest counterexample returned in a counterexample, and the size of Σ , respectively.

For all algorithms discussed, the number of equivalence queries is at most n : each counterexample processed immediately adds at least one new state to the current hypothesis. Note however, that this number is an upper bound and can be expected to vary in practice for the different algorithms. For example, in the discrimination tree algorithm one needs exactly n equivalence queries, since a new state can only be found with such a query. In Angluin's algorithm, on the other hand, the consistency check that is based on membership queries might give rise to new states as well.

The algorithms differ in the number of membership queries. We first discuss the complexity of the algorithm based on observation packs.

In the observation pack algorithm membership queries are performed for two different purposes: to check for closedness and to process a counterexample.

Consider the first type of membership queries. The observation pack is closed when, for every access string s_i and letter a , $s_i a$ is like some other access string. This is easily determined with membership queries. If the check fails, it provides a witness of non-closedness. If it succeeds, a DFA can be built from the answers of the queries.

Each component contains an access string plus at most $n - 1$ strings used to separate it from the other at most $n - 1$ components. Therefore, in the worst case, checking for closedness means asking at most n queries for each of (n) strings s_i and (n) queries for each of (kn) strings $s_i a$, giving a total of $O((k + 1)n^2)$ queries. At this point we can implement the observation pack algorithm in two different ways:

- (1) Check for closedness (and rebuild the automaton) from scratch every time. This means $n(k+1)n^2$ queries.
- (2) Use the fact that access strings are never removed from the pack. This means that the set of queries asked in one closedness check is a subset of the queries to be asked in the next one. So, the total number of different queries over all checks is at most $(k+1)n^2$. We can avoid repeating queries by recording all answers to membership queries, at the expense of using more memory.

Now consider the queries used to process the counterexample. If we do not insist on obtaining the shortest distinguishing experiment, we can use Rivest and Shapire's binary search. This means using $O(\log m)$ queries for each counterexample, hence $O(n \log m)$ for the at most n counterexamples.

In total, the algorithm that records all answers uses at most $O(kn^2 + n \log m)$ membership queries.

This is also the cost of the reduced observation table algorithm, if precisely the data structure recording all answers to membership queries is employed.

The discrimination tree algorithm, as described in [KV94], rebuilds the automaton from scratch every time and processes the counterexample sequentially, so it uses $O(kn^3 + nm)$ membership queries. It is not difficult, however, to make it record previous queries and use binary search to process the counterexample. This modified version will have cost of $O(kn^2 + n \log m)$.

In the observation table algorithm, the number of columns in the table is at most n , but the number of rows can be as large as $O(knm)$ because all prefixes of counterexamples are added as rows. Consequently, the number of queries can be up to $O(kn^2m)$.

It can be shown that for any algorithm, making only $O(n)$ equivalence queries, at least $\Omega(kn \log n)$ membership queries have to be made. Further results on lower bounds can be found in [BDGW97].

Note however, that the results are worst-case estimations. One might in practice trade membership queries for equivalence queries. Experiences with learning algorithms are given in Section 19.4.8.

19.4.7 Domain-Specific Optimizations

The number of queries can be expected to be a limiting factor in practice. Let us study optimizations for learning that are possible when certain further information about the system to learn is provided. The rationale of the presented approach is that in practice, one is often concerned with learning a certain reactive system that can be understood as a special deterministic finite state automaton [HNS03].

The general concept of the optimizations presented here is that instead of the Teacher, an **Assistant** is queried that might either answer a query by consulting the Teacher, or, when possible, deduces the answer to the query using the currently observed information plus the domain specific knowledge about the system to learn.

We present the concept of Assistants using Angluin’s algorithm. It might be transferred to the other learning algorithms in a similar manner. However, since not every algorithm stores the result of all membership queries, the effect might be limited.

The Assistants We present different types of assistants, which differ by the provided context information.

Assistant 1. The first property of reactive systems that we consider is prefix-closedness. If the system enters an error state, it will never recover on further input. So if the system enters a non-accepting state, a sink in the corresponding automaton, it will never leave it. Hence, the automaton’s language is prefix-closed. In other words, prefixes of accepted strings are also accepted and extensions of rejected strings are rejected.

This is used by the first Assistant which states that if a string is a prefix of a string already in \mathcal{OT} with the entry $+$ then the prefix-string will also be entered as $+$, without consulting the Teacher. Similarly, a query for a string that is an extension of a string already classified as rejecting is answered negatively without consulting the Teacher.

Assistant 2. Sometimes, one deals with systems that provide a sequence of output symbols to a given sequence of input symbols. These systems may be modeled as deterministic finite state machines where the input alphabet comprises sequences of input symbols and the output alphabet contains sequences of output symbols. These systems can be understood as DFAs over an alphabet comprising actions that are pairs of sequences of input and output symbols. However, such an alphabet is large and the learning algorithm will be expensive. To eliminate the problem we can split an edge labeled by a sequence of input and output symbols into a sequence of edges where each edge is labeled with a single symbol, first the input symbols and then the output symbols. In this way, the number of states increases but the alphabet is kept small.

Often, these systems are deterministic for a given input. The system under test always produces the same output on any given sequence of inputs. So replacing just one output symbol in a string of an input-deterministic language cannot yield another string of this language. An Assistant can use this knowledge to determine that a membership query should be answered with a $-$ for a certain string if in \mathcal{OT} the same string with the modification of one output symbol has the entry $+$.

Assistant 3. The next Assistant uses the fact that the number of output events in a given situation is determined, and that we wait with feeding new input until the system has produced all its responses. Assume that we have in \mathcal{OT} a string labeled $+$ that ends with an input symbol. Then every string that emerges by changing this input-symbol to any output symbol, will always be rated as $-$. This can be checked by a further Assistant.

Assistant 4. Often, systems are built-up by independent components, executing actions independently. If a and b are such independent actions, an Assis-

tant can deduce that a query $uabv$ to the Teacher will produce the same result as the query $ubav$.

Assistant 5. Furthermore, the system might be built-up by many identical components. Consider there are two identical components A and B of the system. Component A processes the letters a_1, a_2, \dots, a_n whereas component B processes the letters b_1, b_2, \dots, b_n in a symmetrical way. An Assistant which uses this symmetry information can deduce that a query $a_1 b_1 b_2 a_1$ to the Teacher will produce the same result as the query $b_1 a_1 a_2 b_1$.

The concept of Assistants is also used in extensions of these learning algorithms to timed systems [GJL04].

19.4.8 Practical experiences

The presented results on the worst-case complexity of the algorithms introduced gives only limited understanding of their practical performance.

In [BJLS03], Angluin's algorithm has been implemented in a straightforward way in order to gain further insights to practical applicability. Furthermore, its performance on randomly generated automata has been analyzed. The experiments focused on the impact of the alphabet size and the number of states on the needed number of membership queries. Additionally, the optimization for prefix-closed systems mentioned in the previous section (Assistant 1) has been implemented and analyzed.

In general, it turned out that learning is a challenging problem. One obstacle is memory consumption. For example, the observation table for a system of 100 states over 25 letters needed about 160 MB of memory. An arbitrary random system of this size took about 40000 membership queries. A prefix-closed system of the same size required even 110000 queries. In general, it turned out that prefix-closed languages are relatively hard to learn compared to arbitrary regular languages. The optimization, however, showed positive results. Figure 19.14 gives an impression of the number of membership queries needed for learning systems of different sizes.

Further experiences are reported in [HNS03] gained in the process of testing a telecommunication system.

Their experiments have been performed on four finite installations, each consisting of the telephone switch connected to a number of telephones. The systems learned varied in the kind of actions the telephones were able to perform, ranging from simple on-hook and off-hook actions of the receiver to actually performing calls. The output events indicate which actions the telephone switch has performed on the particular input. The assistants mentioned in Section 19.4.7 (called *filters* in [HNS03]) have been employed as well.

In this setup, the automatic execution of a single test needs only a few seconds, but in some exceptional cases it took up to 1.5 minutes to execute the test and to collect the output generated by the system. This is due to the large timeout values that are specified for telecommunication systems. Thus, reducing the number of membership queries has a huge impact.

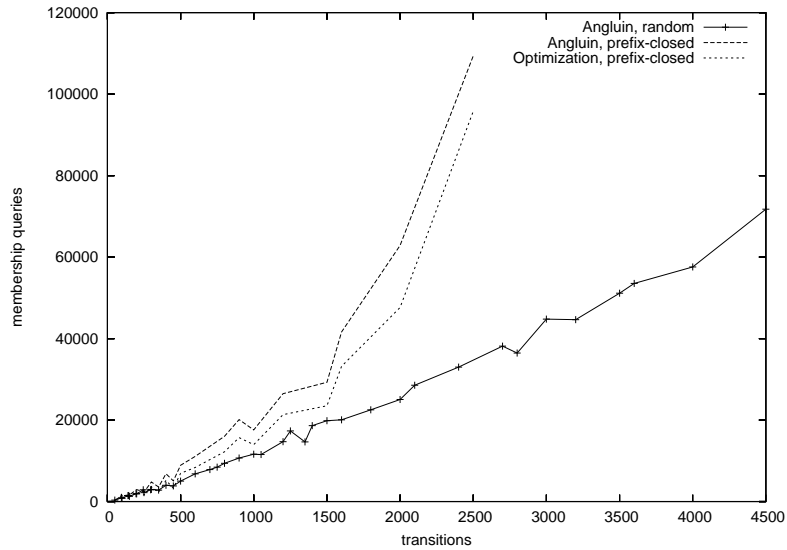


Fig. 19.14. Learning random examples with Angluin’s algorithm

For the measurements, the assistants are used in a cumulative way: First Assistant 1 is used, then Assistant 2 and 3 are added. In the last set of measurements, all assistants have been employed.

The result of adding the assistants are measured in terms of a factor of the number of membership queries saved in comparison to learning the example without any assistants. The factor of reduction varies between 8 to 460 in total when all the assistants are added, depending on the example.

Assistant 1 has a similar impact in all considered scenarios, while Assistant 2, 3, and 4 vary much more in their effectiveness, the saving factor increases with the number of states. The number of outputs and the lengths of output sequences between inputs have a particular high impact on the effects of Assistant 2 and 3. More outputs and longer sequences give a better saving factor.

The impact of Assistant 4 and 5, which covers the partial-order and symmetry aspects, increases, as expected, with the number of independent devices. The number of states does not seem to have any noticeable impact on the effectiveness of these assistants.

19.4.9 Further learning algorithms

Let us assume that we want to create a model of a system that cannot be reset to a start state. Of course, this setting is not meaningful for systems that contain states from which they can never escape once they are entered, since it would not be able to explore the rest of the automaton. Rivest and Schapire [RS93] have created a learning algorithm for systems without reset that are strongly connected.

Moreover Berman and Roos [BR87] present a learning algorithm for a subclass of context-free languages accepted by counter machines and Freund et al. [FKR⁺93] give algorithms for learning finite automata on the basis of a single long walk in an average-case setting. Maler and Pnueli [MP95] study the problem of learning sets of infinite strings. In [DH03a, DH03b], learning of regular tree languages is studied. Learning of timed systems is addressed in [GJL04].

Looking at Angluin’s algorithm, it becomes obvious that there is a trade-off between membership and equivalence queries. Instead of performing an equivalence query for a closed and consistent table, one could compare the row labels of equal rows on further suffixes by membership queries. This might reveal an inconsistency, yielding a separation of the previously equal rows, and thus more states. For every such case, an equivalence query could be saved. This idea is worked out in [BDGW94] and [BGHM96].

19.5 Adaptive Model Checking

In the first section of this chapter, we have studied automatic means for verifying SUTs based on model checking. However, model checking requires a model. If the system under test is a black box, one can use the learning techniques explained in the previous section to learn a model of the box. Then model checking can be applied.

In [GPY02] a method that integrates learning a model of the black box and verifying it is presented. It is termed **Adaptive Model Checking** (AMC). It is similar to the method previously studied in [PVY99] under the term **black box checking**.

Adaptive model checking is a method that deals with the problem of having an inaccurate model of a SUT. Given a property that the system must satisfy, model checking is performed on a preliminary model and if a counterexample is found it is compared with the system under test. The result of the comparison is either that the SUT does not satisfy the property or an automatic refinement of the model.

First, we present an overview of the algorithm in Figure 19.15. The algorithm used for learning is Angluin’s algorithm [Ang87] and the algorithm for performing the equivalence check between the model and the SUT is the Vasilevskii-Chow (VC) algorithm [Vas73, Cho78]. Note that there are two sorts of counterexamples in this setting, videlicet counterexamples produced by the model checker, called mc-counterexamples, and counterexamples produced by the VC algorithm, called vc-counterexamples.

In the black box checking scenario no initial model is assumed to exist and Angluin’s algorithm starts from scratch. The AMC algorithm starts with the model learned so far. This model might be inaccurate. The AMC algorithm applies model checking to this model. There are two possible outcomes of this check:

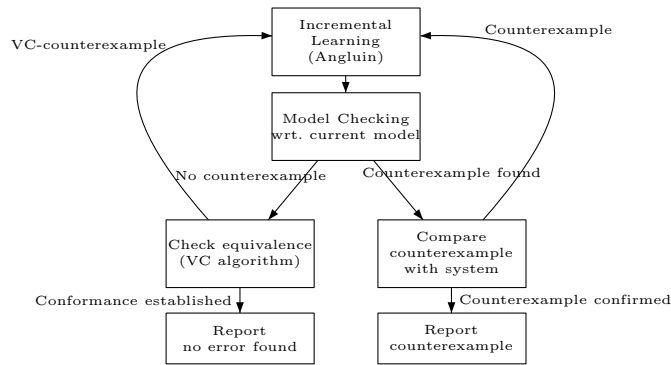


Fig. 19.15. Overview of the Adaptive Model Checking Algorithm.

- (1) If the model checker finds a mc-counterexample for the checked property, the SUT runs the counterexample in order to see if it is indeed a sequence of actions that can be performed by the SUT.
 - If the SUT accepts the mc-counterexample we have an input sequence that shows that the SUT does not satisfy the property. Then the mc-counterexample is reported and the AMC algorithm terminates.
 - In the second scenario where the sequence can not be performed by the SUT, the mc-counterexample will be given to the learning algorithm and the model will be refined.
- (2) In the second case, if the model checker does not produce a mc-counterexample, one has to investigate whether the model corresponds to the SUT. Applying the VC algorithm resolves this question. As before, if a vc-counterexample is found, the counterexample is given to Angluin's algorithm and the model is refined. If no vc-counterexample is found the AMC algorithm concludes that the SUT satisfies the property and the AMC algorithm terminates.

We will now present the AMC algorithm in more detail.

Model The model of the SUT is constructed by Angluin's algorithm in the AMC method. The AMC method assumes that the SUT gives information whether an input can be currently executed by the SUT. Therefore the language of the SUT is assumed to be prefix-closed and hence the model as well. The model is a finite automaton and its runs represent only the successful experiments (strings) in the SUT. The learning algorithm used in AMC to learn this model is Angluin's algorithm. We assume that we know an upper bound on the number of states, n , of the SUT.

Property The property is given as an LTL formula, and this can be translated into a Büchi automaton. So the property can be expressed in Linear Temporal

Logic (see Section 19.3.2 on Linear Temporal Logic) and transformed into a finite automaton $A_{\neg\phi}$ on infinite words not accepting the property, usually a Büchi automaton. The algorithm for model checking LTL is presented in detail in Section 19.3.3. LTL is used since it delivers counterexamples in the format of a sequence of actions. Since Computational Tree Logic is not suitable to describe properties of regular languages and counterexamples in CTL are not just words of a language, CTL cannot be used in the AMC algorithm.

Initialization Let \mathcal{M} be the SUT and \mathcal{A}_{init} an initial model of \mathcal{M} in the following section. The AMC algorithm is initialized by providing Angluin’s algorithm with the information of the initial model so that fewer calls to the (time expensive) VC algorithm are needed. In [GPY02], three ways to use the initial model are proposed.

- (1) A false negative mc-counterexample t found (i.e., a sequence t that was considered to be a counterexample, but has turned out not to be an actual execution of the SUT \mathcal{M}). This corresponds to a counterexample in Angluin’s algorithm. Following Angluin’s algorithm we perform membership queries for all prefixes of t .
- (2) The runs T of a spanning tree of the model \mathcal{A}_{init} as the initial set of row labels S_A (access strings). We initialize Angluin’s algorithm by adding each $s \in S_A$ to \mathcal{OT} and perform membership queries for the missing entries.
- (3) A set of separating sequences $DS(\mathcal{A}_{init})$ calculated for the states of \mathcal{A}_{init} as the initial set of column labels E_A . Thus, we initialize Angluin’s algorithm by setting \mathcal{OT} to be empty, and $E_A = DS(\mathcal{A}_{init})$.

So this three ways of initializing the sets S_A and E_A account for the attempt to speed up learning, but with the entries in \mathcal{OT} queried for on the SUT at hand now.

Note that when using all three initializations and if \mathcal{A}_{init} models \mathcal{M} accurately with these choices of S_A and E_A then this will allow Angluin’s algorithm to learn \mathcal{A}_{init} correctly, without the assistance of the expensive equivalence check (VC algorithm).

Handling model-checking counterexamples The model checker can construct two types of mc-counterexamples to a LTL formula, finite and infinite. In the first case there is no extra work in handling it: The counterexample is given directly to Angluin’s algorithm. In the second case we must make the counterexample finite in order for Angluin’s algorithm to use it.

An infinite counterexample is an ultimately periodic word of the form $w_1 w_2^\omega$ where $w_1, w_2 \in \Sigma^*$. Assuming that the automaton being checked has n number of states, the counterexample given to Angluin’s algorithm is $w_1 w_2^{n+1}$. Running w_1 in $A_{\neg\phi}$ it needs to terminate in an accepting state s . Starting from s the second part w_2 needs to terminate in s as well. For each such pair, we apply the second part n more times. That is, we try to run the string $w_1 w_2^{n+1}$. If we succeed, this means that there is a cycle in $A_{\neg\phi} \cap \mathcal{M}$ through a state with s

as the $A_{\neg\phi}$ component. This is the case since there are at most n ways to pair up s with a state of \mathcal{M} . In this case, there is an infinite accepting path in the intersection.

Experiments and discussion In [GPY02], an experimental implementation of the algorithm is analyzed. Two CCS models [Mil89] are learned and two properties are checked. The two selected properties do not hold, and the correct models are tampered with in order to experiment with finding a (false negative) mc-counterexample. This counterexample is in all experiments, using the AMC method, utilized to initialize the observation table in Angluin’s algorithm. This is the initialization mentioned in Section Initialization, case 1. The model checker checks the properties sequentially. The experiments have been performed on SUTs with approximately 500 and 100 states, respectively.

The experiments aim to compare the black box checking method (in which Angluin’s observation table is not initialized) and the AMC method. Experiments are performed in the AMC method with different combinations to initialize Angluin’s observation table, choices (1) and (2) (then $E_A = \{\varepsilon\}$), (1) and (3) (then $S_A = \{\varepsilon\}$) and (1), (2), and (3) are used.

The results can be summarized as follows. Comparing the different ways of initializing \mathcal{OT} we see that learning from scratch are in these experiments slower than using initialized tables. The results also indicate that AMC method give rise to more states in the model than the black box checking method. Furthermore, the counterexamples are shorter when learning from scratch than those when initialized tables are used, if the number of states of the initialized tables are large.

The adaptive model checking methods is applicable for models that are inaccurate but not completely irrelevant. When comparing an algorithm learning from scratch, and using an initial model to guide the learning of the modified SUT (AMC) the different benefits over each other are unveiled. The learning from scratch method can be useful when there is a short error trace that identifies why the checked property does not hold. In this case, it is possible that the learning from scratch method will discover the error after learning only a small model. The AMC method is useful when the modification of the SUT is simple or when it may have a very limited affect on the correctness of the property checked.

However, it has to be said that adaptive model checking is still a mainly unexplored area that further theory as well as practical insights are needed.

19.6 Summary

In this chapter an introduction to model checking and model learning was given. Furthermore, it was shown how to combine both techniques to an approach in which properties of a SUT are verified directly.

First of all we have presented Kripke transition systems which build a simple basis for temporal logics used in model checking. The essential difference between

linear time logics and branching time logics was made plain on the basis of an example. Subsequently we presented *linear time logic* (LTL) and computational tree logic (CTL) which are widely used for model checking purposes. Since the combination of model checking and model learning for testing purposes is only meaningful with linear time logics we presented a basic model checking algorithm for linear time logic.

In the second part of the chapter we first gave an introduction to the general ideas of model learning algorithms. Continuing in the same subject, we presented a number of learning algorithms; the observation pack algorithm, Angluin's algorithm, the reduced observation table algorithm and, the discrimination tree algorithm. Subsequently we discussed the algorithms' query complexity and presented some domain specific optimizations to reduce the number of queries. We rounded the model learning part off with some experimental results.

The final part in this chapter presented the adaptive model checking algorithm, which combines model checking and model learning into one approach. The approach try to make use of information in an existing model of the SUT in order to save effort in the learning procedure. If no model exist or the existing model is irrelevant compared to the current SUT, the approach is still applicable.

Although model checking and model learning are both established research areas, a lot of work remains to be done when considering testing. The combination of model checking and testing techniques should be clarified. Models to be used for testing might ask for different characteristics of the learning procedures than they currently have. For example, the construction of an abstract model of a SUT using learning algorithms might ask for a new approach. Issues in this area need to be examined from a theoretical as well as practical point of view.

Literature

- [Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987. [28, 40, 45]
- [BCL92] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proceedings of the International Conference on Very Large Scale Integration (VLSI 1991)*, volume A-1 of *IFIP Transactions*, pages 49–58, Edinburgh, Scotland, 1992. North-Holland. [13]
- [BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Conference on Design Automation Conference (DAC 1990)*, pages 46–51. ACM Press, 1990. [13]
- [BCMS01] O. Burkart, D. Caucal, F. Moller, and Bernhard Steffen. Verification on infinite structures. In S. Smolka J. Bergstra, A. Pons, editor, *Handbook on Process Algebra*. North-Holland, 2001. [6]
- [BDGW94] J. L. Balcázar, J. Díaz, R. Gavaldà, and O. Watanabe. The query complexity of learning DFA. *New Generation Computing*, 12:337–358, 1994. [45]
- [BDGW97] J. L. Balcázar, J. Díaz, R. Gavaldà, and O. Watanabe. Algorithms for learning finite automata from queries: A unified view. In Ding-Zhu Du, Ker-I Ko, and Dingzhu Du, editors, *Advances in Algorithms, Languages, and Complexity*. Kluwer Academic, February 1997. In Honor of Ronald V. Book. [23, 36, 41]
- [BGHM96] N. H. Bshouty, S. A. Goldman, T. R. Hancock, and S. Matar. Asking queries to minimize errors. *Journal of Computer and Systems Science*, 52:268–286, 1996. [45]
- [BJLS03] Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena. Insights to Angluin’s learning. In *Proceedings of the International Workshop on Software Verification and Validation (SVV 2003)*, Electronic Notes in Theoretical Computer Science, December 2003. To appear. [43]
- [BR87] P. Berman and R. Roos. Learning one-counter languages in polynomial time. In *Proceedings of the 28th IEEE Symposium on the Foundations of Computer Science (FOCS 1987)*, pages 61–67, Los Alamitos, CA, 1987. IEEE Computer Society Press. [45]
- [Büc62] J. R. Büchi. On a decision method in restricted second-order arithmetic. In E. Nagel, P. Suppes, and A. Tarski, editors, *Proceedings of the 1st International Congress for Logic, Methodology, and Philosophy of Science (LMPS 1960)*, pages 1–12. Stanford University Press, 1962. [5, 13]
- [CCG⁺02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An open-source tool for symbolic model checking. In E. Brinksma and K. Guldstrand Larsen, editors, *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364, Copenhagen, Denmark, July 2002. Springer-Verlag. [21]
- [Cho78] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, May 1978. Spe-

- cial collection based on the 2nd International Computer Software and Applications Conference (COMPSAC 1978). [39, 45]
- [CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based verification tool for finite state systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(1):36–72, January 1993. [21]
- [CS92] Rance Cleaveland and Bernhard Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. In Kim G. Larsen and Arne Skou, editors, *Proceedings of the 3rd Conference on Computer Aided Verification (CAV 1991)*, volume 575, pages 48–58, Berlin, Germany, 1992. Springer-Verlag. [12]
- [DH03a] F. Drewes and J. Högberg. Learning a regular tree language from a teacher. In Z. Ésik and Z. Fülöp, editors, *Proceedings of the 7th International Conference on Developments in Language Theory (DLT 2003)*, volume 2710 of *Lecture Notes in Computer Science*, pages 279–291. Springer-Verlag, 2003. [45]
- [DH03b] F. Drewes and J. Högberg. Learning a regular tree language from a teacher even more efficiently. Technical Report 03.11, Umeå University, 2003. [45]
- [EH00] K. Etessami and G. J. Holzmann. Optimizing Büchi automata. In C. Palamidessi, editor, *Proceedings of 11th International Conference on Concurrency Theory (CONCUR 2000)*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–167. Springer-Verlag, 2000. [13]
- [EL85] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking (extended abstract): Branching time strikes back. In *Conference Record of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1985)*, pages 84–96. ACM Press, 1985. [11]
- [FGM⁺92] J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A toolbox for the verification of LOTOS programs. In *Proceedings of the 14th International Conference on Software Engineering (ICSE 1992)*, pages 246–259. ACM Press, 1992. [21]
- [FKR⁺93] Y. Freund, M. Kearns, D. Ron, R. Rubinfeld, R. Schapire, and L. Sellie. Efficient learning of typical finite automata from random walks. In *Proceedings of the 25th ACM Symposium on the Theory of Computing (STOC 1993)*, pages 315–324, New York, NY, 1993. ACM Press. [45]
- [GJL04] O. Grinchtein, B. Jonsson, and M. Leucker. Learning of event-recording automata. Technical report, Uppsala University, 2004. [43, 45]
- [GO01] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer-Verlag, 2001. [13]
- [GPY02] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In J.-P. Katoen and P. Stevens, editors, *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *Lecture Notes in Computer Science*, pages 357–370. Springer-Verlag, 2002. [45, 47, 48]
- [HHK96] R. H. Hardin, Zvi Har’El, and Robert P. Kurshan. COSPAN. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification (CAV 1996)*, volume 1102 of *Lecture Notes in Computer Science*, pages 423–427. Springer-Verlag, 1996. [21]

- [HK87] Z. Har'El and R. P. Kurshan. *COSPAN User Guide*. AT&T Bell Laboratories, October 1987. [21]
- [HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985. [11]
- [HNS03] H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In W. A. Hunt Jr. and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003)*, Lecture Notes in Computer Science, pages 315–327. Springer-Verlag, 2003. [41, 43]
- [Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. [21]
- [Koz83] D. Kozen. Results on the propositional μ -calculus. *Theoretical Computer Science*, 27:333–354, December 1983. [12]
- [KV94] M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory*. The MIT Press, Cambridge, Massachusetts and London, England, 1994. [36, 41]
- [Mil89] Robin Milner. *Communication and concurrency*. Prentice-Hall, 1989. [11, 48]
- [MOSS99] M. Müller-Olm, D. Schmidt, and B. Steffen. Model checking: A tutorial introduction. In G. File A. Cortesi, editor, *Proceedings of the 6th Static Analysis Symposium (SAS'99)*, volume 1694 of *Lecture Notes in Computer Science*, pages 330–354, Heidelberg, Germany, September 1999. Springer-Verlag. [6]
- [MP95] O. Maler and A. Pnueli. On the learnability of infinitary regular sets. *Information and Computation*, 118(2):316–326, 1 May 1995. [45]
- [Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium Foundations of Computer Science (FOCS 1977)*, pages 46–57. IEEE Computer Society Press, 1977. [8]
- [PVY99] D. Peled, M. Vardi, and M. Yannakakis. Black box checking. In *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE 1999) and Protocol Specification, Testing and Verification (PSTV 1999)*, volume 156 of *IFIP Conference Proceedings*, pages 225–240. Kluwer Academic, 1999. [45]
- [RS93] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, April 1993. [32, 44]
- [SCK⁺95] B. Steffen, A. Claßen, M. Klein, J. Knoop, and T. Margaria. The fixpoint analysis machine. In J. Lee and S. Smolka, editors, *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR 1995)*, volume 962 of *Lecture Notes in Computer Science*, pages 72–87, Heidelberg, Germany, 1995. Springer-Verlag. [21]
- [SVW87] A. P. Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for Büchi automata with application to temporal logics. *Theoretical Computer Science*, 49:217–237, 1987. [13]
- [SW91] Colin Stirling and David Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177, October 1991. [12]
- [Var96] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. M. Birtwistle, editors, *Logics for Concurrency – Structure versus Automata. Proceedings of the 8th Banff Higher Order Workshop*

- (*Banff 1995*), volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag, 1996. [12, 13]
- [Var01] M. Y. Vardi. Branching vs. linear time: Final showdown. In W. Yi T. Margaria, editor, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer-Verlag, January 2001. [8]
- [Vas73] M. P. Vasilevski. Failure diagnosis of automata. *Cybernetic*, 9(4):653–665, 1973. [39, 45]
- [WVS83] P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasoning about infinite computations paths. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science (FOCS 1983)*, pages 185–194. IEEE Computer Society Press, 1983. Extended abstract. [13]

Paper B

Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena. Insights to Angluin's learning. *Electronic Notes in Theoretical Computer Science*, 118:3–18, 2005. Reprinted with permission from Elsevier.

Insights to Angluin’s Learning

Therese Berg¹ Bengt Jonsson¹ Martin Leucker,^{1,2}
Mayank Saxena¹

*Department of Computer Systems
Uppsala University
Uppsala, Sweden*

Abstract

Among other domains, learning finite-state machines is important for obtaining a model of a system under development, so that powerful formal methods such as model checking can be applied.

A prominent algorithm for learning such devices was developed by Angluin. We have implemented this algorithm in a straightforward way to gain further insights to practical applicability. Furthermore, we have analyzed its performance on randomly generated as well as real-world examples. Our experiments focus on the impact of the alphabet size and the number of states on the needed number of membership queries. Additionally, we have implemented and analyzed an optimized version for learning prefix-closed regular languages. Memory consumption is one major obstacle when we attempted to learn large examples.

We see that prefix-closed languages are relatively hard to learn compared to arbitrary regular languages. The optimization, however, shows positive results.

Key words: deterministic finite-state automata, learning
algorithm, regular languages, prefix-closed regular languages

1 Introduction

The last decades have witnessed significant advances in *model-based techniques* for specification, implementation, verification, and validation of reactive and usually distributed systems, e.g., in telecommunication, embedded control, and related application areas. The techniques include model checking [11,5], code generation [8] and model-based test generation [4,12]. They all assume that a formal model of the system under study is available. Such formal

¹ E-mail addresses: thereseb@docs.uu.se, bengt@docs.uu.se, leucker@docs.uu.se, mayanks@docs.uu.se

² This author is supported by the European Research Training Network “Games”.

models are assumed to be developed during the specification phase of system development, or *a posteriori* from an existing implementation.

One large obstacle to the adoption of model-based techniques is that in practice, quite often *no* formal specification is available or it is *outdated* due to the iteration process in the development of a system. Even if a formal specification is present that captures the latest version of the system intended to develop, it is not clear whether it corresponds to its actual realization.

One approach to overcome these limitations is to develop techniques for generating formal models with less manual effort and more automated support. In the extreme case, a formal model could be generated *a posteriori*, from the developed system. If no model of the system under development was present, this model can be used to analyze and validate the implementation. If a formal model was available *a priori*, the generated model can be compared with this one to show conformance of the implementation with respect to its specification.

For software systems with given source code, various static and dynamic analysis techniques have been developed, which can also be used to generate abstract models of a developed system [3,9]. However, peripheral hardware systems, combined hard- and software systems, or third-party software systems do not allow means of static analysis. In practice, there is often no other way to analyze these systems than by looking at their traces, i.e., their sequences of input and output actions. Also, a program that analyzes the source code statically is heavily dependent on the particular implementation language used. A tool that analyzes externally observed traces is easier to adapt to a new program written in a new language.

In a seminal paper, Angluin [1] described a method for learning finite-state automata, if it is possible to ask whether a string is a member of the language of the automata. This result implies that, in principle, finite-state automata can be learned for finite-state systems that have the following two characteristics:

- one can send sequences of actions to the system and
- the system signals whether it could execute the sequence.

This approach has been used in projects for test sequence generation by Steffen et al. [7], and by Peled et al. for developing techniques for conformance testing of finite automata [6]. The number of reported efforts to use Angluin's algorithm (or some related algorithm) for generating finite automata models of reactive systems is still rather small and it is still not possible to make conclusions about the applicability of the techniques, how well it scales, or to pinpoint the crucial bottlenecks.

The objective of the research reported in this project is to investigate the efficiency of Angluin's algorithm for learning finite automata, and among them models of reactive systems, to investigate potential bottlenecks in applying it, and to investigate the effect of a rather straight-forward optimization for prefix-closed DFA. For this purpose, we have developed a naive implemen-

tation of Angluin’s algorithm together with an optimization, which can optionally be invoked. We have applied this implementation to a series of synthetically generated systems, and to a set of rather simple models of reactive systems intended for verification by the Concurrency Workbench. From the results, we draw conclusions regarding the applicability and scalability of Angluin’s algorithm, as well as the effect of our implemented optimization.

[10] studies domain-specific optimizations to Angluin’s learning algorithm including optimizations for prefix-closed languages. They have considered examples from telecommunication software but not studied the performance on synthetic examples. They have in this article used a slightly different model and therefore we could not easily compare our results. In [2], Angluin revisits his algorithm and discusses several variants and their complexity. Practical results, however, are not mentioned.

In the next section, we recall basic definition of automata theory. In Section 3, we describe Angluin’s learning algorithm as well as our optimization for prefix-closed languages. Our experiments are described and discussed in Section 4.

2 Preliminaries

For the following, we fix an *alphabet* Σ , i.e. a finite set of *letters*, usually denoted by $a, b, \dots, a_1, a_2, \dots$. A *language* is a subset of Σ^* , the set of finite (possibly empty) sequences of letters, also called *strings* or *words*.

A *deterministic finite-state automaton (DFA)* over Σ is a structure $\mathcal{A} = (Q, \delta, q_0, F)$ where Q is a non-empty finite set of *states*, $q_0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states*, and $\delta : Q \times \Sigma \rightarrow Q$ is the *transition function*. We denote the number of states Q , the size of the alphabet Σ , and the size of the transition function δ by respectively $|Q|$, $|\Sigma|$, and $|\delta|$. The latter is defined to be the number of elements of the domain of δ , i.e. $|Q \times \Sigma|$.

A *run* of \mathcal{A} on a finite word $w = a_1 \dots a_n \in \Sigma^*$ is a sequence $q_0 \xrightarrow{a_1} \dots \xrightarrow{a_n} q_n$, where q_0 is the initial state of \mathcal{A} and $q_{i+1} = \delta(q_i, a_{i+1})$ for $i \in \{0, \dots, n-1\}$. It is called *accepting*, if $q_n \in F$. The *language* accepted by \mathcal{A} , denoted by $\mathcal{L}(\mathcal{A})$, is defined as $\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \text{there is an accepting run of } \mathcal{A} \text{ on } w\}$. We call a language \mathcal{L} *regular* if there is a DFA accepting \mathcal{L} .

Let us recall the notion of Nerode’s right congruence. Given a language \mathcal{L} , we say that two words $u, v \in \Sigma^*$ are *equivalent*, written as $u \equiv_{\mathcal{L}} v$, if for all $w \in \Sigma^*$ we have $uw \in \mathcal{L}$ iff $vw \in \mathcal{L}$. It is easy to see that $\equiv_{\mathcal{L}} \subseteq \Sigma^* \times \Sigma^*$ is a right congruence, i.e., it is an equivalence relation that additionally satisfies $u \equiv_{\mathcal{L}} v$ implies $uw \equiv_{\mathcal{L}} vw$ for all $w \in \Sigma^*$. We denote the equivalence class of a word w by $[w]$.

It is folklore that a language \mathcal{L} is regular iff $\equiv_{\mathcal{L}}$ has finite *index*, i.e., the number of equivalence classes of Σ^* with respect to $\equiv_{\mathcal{L}}$ is finite. Let us recall the idea of the proof for the direction *right-to-left*: Given a language \mathcal{L} with finite index, we construct an automaton $\mathcal{A}_{\mathcal{L}}$ such that $\mathcal{L}(\mathcal{A}_{\mathcal{L}}) = \mathcal{L}$. The states of $\mathcal{A}_{\mathcal{L}}$ are the equivalence classes of Σ^* with respect to $\equiv_{\mathcal{L}}$, the initial state is

the equivalence class containing the empty string, denoted by ε , final states are the ones containing strings in \mathcal{L} , and the transition function maps $([w], a)$ to $[wa]$.

It can be shown that this construction yields a minimal DFA accepting \mathcal{L} , i.e., the number of states is minimal among all DFA accepting \mathcal{L} . Furthermore, it can be shown that every minimal DFA is isomorphic to the one we constructed.

3 Learning finite state machines

3.1 Angluin’s learning algorithm

We here try to give a succinct description of the main ideas behind Angluin’s learning algorithm. We assume that a system in which we are interested can be modeled by a DFA \mathcal{A} . The problem can now be looked upon as identifying the regular language which is accepted by \mathcal{A} , denoted by $\mathcal{L}(\mathcal{A})$.

In a learning algorithm a so called *Learner*, who initially knows nothing about \mathcal{A} , is trying to learn $\mathcal{L}(\mathcal{A})$ by asking queries to a *Teacher* and an *Oracle*. There are two kinds of queries.

- A *membership query* consists in asking the *Teacher* whether a string $w \in \Sigma^*$ is in $\mathcal{L}(\mathcal{A})$.
- An *equivalence query* consists in asking the *Oracle* whether a hypothesized DFA \mathcal{M} is correct, i.e., whether $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{A})$. The *Oracle* will answer *yes* if \mathcal{M} is correct, or else supply a counterexample u , either in $\mathcal{L}(\mathcal{A}) \setminus \mathcal{L}(\mathcal{M})$ or in $\mathcal{L}(\mathcal{M}) \setminus \mathcal{L}(\mathcal{A})$.

The typical behavior of a *Learner* is to start by asking a sequence of membership queries, and gradually build a hypothesized DFA \mathcal{M} using the obtained answers. When the *Learner* feels that she has built a “stable” hypothesis \mathcal{M} , she makes an equivalence query to find out whether \mathcal{M} is correct. If the result is successful, the *Learner* has succeeded, otherwise she uses the returned counterexample to revise \mathcal{M} and perform subsequent membership queries until arriving at a new hypothesized DFA, etc.

The information gained by the *Learner* can during the learning process be represented as a partial mapping \mathcal{O} from Σ^* to $\{\textit{accepted}, \textit{rejected}\}$. The domain $\textit{Dom}(\mathcal{O})$ of \mathcal{O} is the set of strings for which membership queries have been performed, or which the *Oracle* has given as counterexamples in equivalence queries.

Roughly speaking, a learning algorithm should prescribe how to transform a partial mapping \mathcal{O} into an automaton. This can be done by fixing a subset S of $\textit{Dom}(\mathcal{O})$, defining an equivalence relation \simeq on S , and building the automaton as the set of equivalence classes of strings in S . Intuitively, two strings u and u' should be equivalent if there is reason to believe that $u \equiv_{\mathcal{L}(\mathcal{A})} u'$. Since the *Learner* can only obtain partial information about \mathcal{A} from \mathcal{O} , one idea is to approximate $\equiv_{\mathcal{L}(\mathcal{A})}$ by an equivalence \simeq , which uses only information in \mathcal{O} .

To be able to build an automaton on the basis of S and \simeq , the following two criteria should preferably be satisfied.

- *completeness*: If $u \in S$, and $a \in \Sigma$ then $ua \simeq u'$ for some $u' \in S$.
- *consistency*: If $u \simeq u'$ for $u, u' \in S$ and $a \in \Sigma$, then $ua \simeq u'a$ (i.e., \simeq is a right congruence).

Completeness ensures that we can define transitions from each equivalence class for each letter in Σ ; consistency ensures that such transitions have a unique target equivalence class. We note that in order to check completeness and consistency, it is necessary to define \simeq on all strings u and ua such that $u \in S$ and $a \in \Sigma$. Whenever the current values of S and \simeq satisfy the completeness and consistency criteria, the *Learner* can form the corresponding hypothesis \mathcal{M} and make an equivalence query about \mathcal{M} .

Let us now describe Angluin's algorithm more specifically. Angluin's algorithm maintains a prefix-closed set S and a suffix-closed set E of strings, both of which are monotonically increased during the algorithm. Initially S and E contain the empty string ε . We define \simeq as follows: $u \simeq v$ if for all $w \in E$ we have $uw \in \mathcal{L}(\mathcal{A})$ iff $vw \in \mathcal{L}(\mathcal{A})$.

From a complete and consistent \mathcal{O} , a hypothesis \mathcal{M} is formed as the automaton, whose states are equivalence classes of strings in S . If \mathcal{O} is not complete, then S is increased with strings that represent missing equivalence classes. If \mathcal{O} is not consistent, E is increased with a suffix which replaces the inconsistent equivalence class with two new classes.

The description of Angluin's algorithm in [1] represents \mathcal{O} by an *observation table* \mathcal{OT} . The observation table is a table with rows corresponding to strings in S and columns corresponding to strings in E . The algorithm gradually fills the entry (u, v) for row u and column v by *accepted* or *rejected*, after receiving a reply for a membership query for uv .

Of course, some membership queries can be saved by entering also the counterexamples returned in negative equivalence queries as accepted or rejected. We note that the observation table is redundant in that the result of a membership query for u occurs in all entries (v, w) such that $u = vw$. Thus, we do not need to make one membership query for each such entry, but we can simultaneously fill all such entries.

Angluin's algorithm is designed to construct minimal DFA for the guessed language.

3.2 Prefix-closed models

In many applications, we want to learn an automaton \mathcal{A} , which is a model of a reactive system. Often, reactive systems can be modeled as transition systems. These can be understood as (non-deterministic) finite-state automata (with partial transition relations) in which every state is an accepting state. Thus, the language defined by such an automaton will be prefix-closed. In this

section, we discuss how to exploit this fact for optimizing the learning process. A language \mathcal{L} is *prefix-closed* if for every w in \mathcal{L} , all prefixes of w are in \mathcal{L} . A DFA is *prefix-closed* if its language is prefix-closed. It follows that a minimal prefix-closed DFA has only one non-final state, the so-called *sink*, with transitions only to itself. Note that Angluin’s algorithm learns minimal DFA.

Studying strings that are possibly accepted by prefix-closed DFA, we make the following simple but important observations:

- (i) Prefixes of accepted strings are accepted.
- (ii) Extensions of rejected strings are rejected.

We can use these characteristics of prefix-closed DFA to reduce the needed number of membership queries as follows. Before querying a string, we first test it for (ii), that is whether it is an extension of a string already observed to be rejected. If so, we can add the result immediately to the observation table. Otherwise, we ask the teacher. Thus, we never need to query extensions of observed rejected strings.

Angluin’s *Learner* starts with queries for short strings, and thereafter queries successively longer and longer strings. In general, the test for (i) will not be able to consult previous observations, so it is rarely applicable. There is, though, an exception when it could be useful, namely when performing queries for prefixes of received counterexamples. If the counterexample c is accepted, we know that all its prefixes are accepted, too. In the best case, applying (i) would save me membership queries, where m is the maximum length of any received counterexample and e is the number of equivalence queries made. Knowing the best case bound and due to lack of evaluation time, we did not implement (i).

4 Experimental Results

The implementation

We have implemented Angluin’s learning algorithm, closely following the high-level description in [1]. Furthermore, we have implemented our proposed optimization for prefix-closed models. It differs from the ordinary learner only in that it can infer the answer of some membership queries, due to properties of prefix-closed languages. Accordingly, the number of membership queries can be expected to be smaller, while the number of equivalence queries is unchanged. Furthermore, it requires the same amount of memory for the observation table.

We simulated the teacher (and oracle) on the same computer as the learner, for reasons of simplicity. In practice, a teacher will typically be realized as a process communicating with a slow external device.

Our learners are written in Java using the library AMoRE developed at RWTH Aachen for maintaining automata.

The experiments

Our experiments aim at finding out how our implementation of Angluin’s learner and our optimized learner perform and scale in practice. We have examined real-world examples and randomly generated examples. The real-world examples are several protocols shipped with the Edinburgh Concurrency Workbench. They were originally formulated in Milner’s CCS. We transformed their transition system representation into minimized prefix-closed DFA.

For reasons of comparison, we studied two kinds of random examples, prefix-closed random examples and arbitrary random DFA.

As pointed out before, the expected bottle-neck in practice for a learner is the number of membership and equivalence queries, since a communication with a typically slow external device is required and quite many queries are needed. Thus, we concentrated our experiments on the number of membership and equivalence queries. To get an overall picture, we also measured the execution time and memory consumption for large examples. Hereby “execution time” means the total execution time minus the time for equivalence queries. In other words, we measure the time spent by the learner plus the one spent by the teacher. Since there are several ways to realize The oracle, we disregard this time.

In our experiments, we vary the alphabet size and the number of states of the automata. Our measurements do not adhere to strong statistical requirements. Thus, they cannot be used to *prove* the practical performance of the algorithms in a statistical sense. However, they are good enough to show a tendency and to point out directions for future optimizations and analysis.

4.1 Angluin’s algorithm

A theoretical upper bound for the number of membership queries is the worst-case size of the observation table. In, [1], Angluin calculates this bound to $O(m|\Sigma||Q|^2)$, where m is the maximum length of any received counterexample. If the *Oracle* always provides a smallest counterexample, then $m = |Q|$, and thus the number of membership queries are in the worst case $O(|\Sigma||Q|^3)$.

To investigate how the algorithm behaves in practice, we studied it on arbitrary random examples as well as prefix-closed random examples. Let us start with the arbitrary ones.

4.1.1 Random examples

The samples

We studied random examples varying the number of states and letters. We generated and learned DFA between 10 and 100 states, in steps of 10.

Each set of measurements was carried out with different alphabet size. For systems with up to 60 states, we studied from 5 up to 50 letters in steps of 5, and, for systems with more states, from 10 up to 50 letters in steps of 10.

We sampled 10 DFA for each state and letter combination, except for those

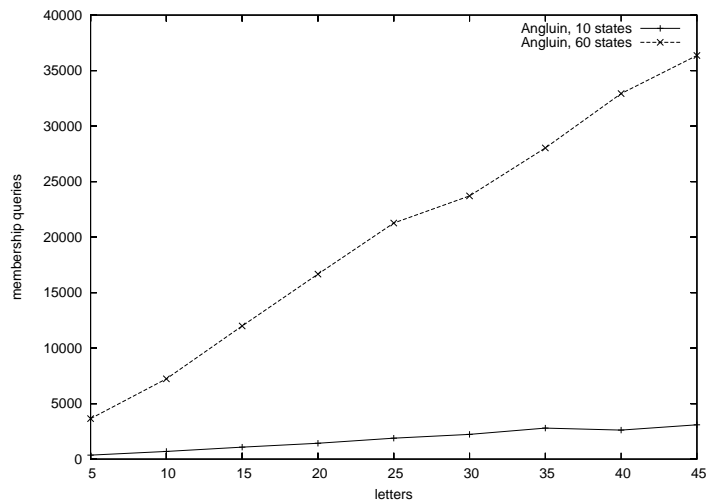


Fig. 1. Random automata, number of states fixed to 10 and 60.

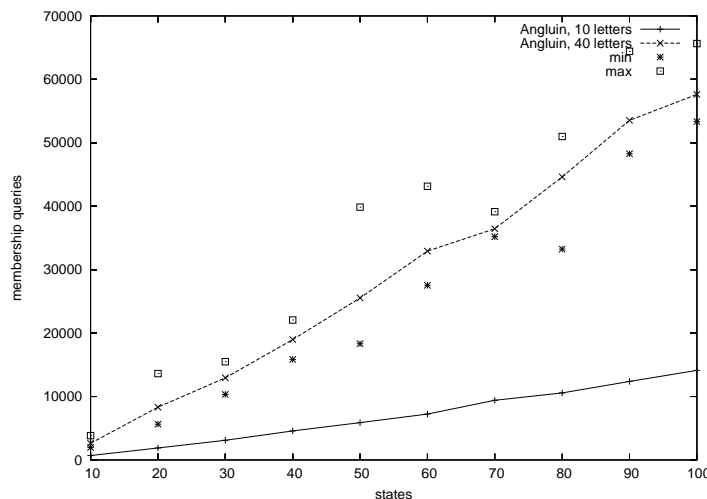


Fig. 2. Random automata, number of letters fixed to 10 and 40.

with the number of states 70 or higher, for which we sampled only 5.

Experiences

Fixing the number of states but varying the number of letters, we observe a linear behavior, as expected. See Figure 1, in which the number of states are fixed to 10 and 60.

The collected data shows that, in terms of membership queries, Angluin’s learning algorithm is approximately linear in states on random DFA, despite the algorithm’s worst-case complexity. As an example, we show the number of membership queries relative to the number of states, with the number of letters fixed to 10 and 40, in Figure 2.

To get an impression of the performance of the algorithm, learning a random example of 100 states and 25 letters, took 1 hour, 40,000 membership and 15

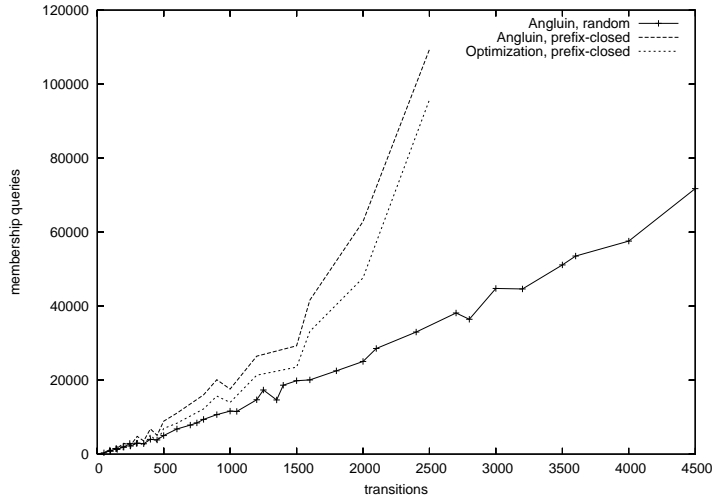


Fig. 3. Membership queries with respect to transitions.

equivalence queries, and 110 MB of space. This long execution time was one reason for learning fewer automata of larger sizes. The other reason is the huge memory consumption of the observation table.

Additionally, we studied the number of membership queries with respect to the number of transitions, $|\delta| = |Q||\Sigma|$. This is possible since we discovered by our measurements that the nominal variables states and letters behave interchangeably. At the very bottom in Figure 3 is the curve that shows the number of membership queries with respect to the number of transitions. We can describe the observations roughly by the relation $|membership\ queries| = k|\delta|$, where $k \approx 14$.

4.1.2 Random prefix-closed examples

The samples

In general, prefix-closed DFA require more time and space to learn with Angluin’s algorithm, so we studied fewer samples. We learned automata with 10 to 50 states in steps of 10 and varied the number of letters from 10 to 50 in steps of 10. We learned approximately 10 automata of each kind up to 30 states and fewer of larger ones.

Experiences

As mentioned before, arbitrary random examples are in general easier to learn than prefix-closed random examples. An example for this is shown in Figure 4. Learning a particular random generated automaton, with 40 letters and 40 states, requires approximately 19,000 membership queries and a prefix-closed automaton of the same size requires 40,000 membership queries, that is about twice as many.

Let us study the cause for this difference. Angluin’s learning algorithm tries to learn an automaton by finding representatives of different Nerode’s right

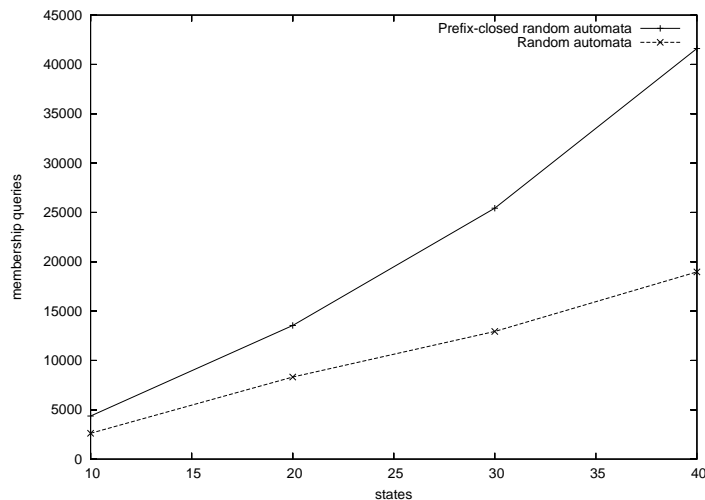


Fig. 4. Random prefix-closed automata and random automata, both with 40 letters.

congruence classes, as described in Section 3.1. To show that two strings u and v are not members of the same congruence class, it has to find one string w so that uw is accepted but vw not, or the other way around. In minimal prefix-closed DFA, as maintained by Angluin, every state except one is accepting, and it is more likely that states accept almost the same language. This makes it more difficult to find a distinguishing string w .

Furthermore, we see that the curves for learning prefix-closed languages grow steeper than for arbitrary random examples. On prefix-closed examples we come closer to the worst case complexity of Angluin’s algorithm. Thus, prefix-closed examples are harder to learn than arbitrary ones. This result is slightly disappointing, since reactive systems can usually be modeled by prefix-closed automata. This experience is in contrast to the one gained in the area of model checking, where worst-case complexities usually do not show up in real-world examples.

A particular random example with 100 states and 25 letters took 11 hours, 110,000 membership queries, 29 equivalence queries and 160 MB of memory. The top curve in Figure 3 shows membership queries versus transitions for random prefix-closed examples. It is no longer linear. A very rough description of the observations is given by the quadratic relation $|membership\ queries| = k|\delta|^2$, where $k \approx 0.016$.

4.2 The optimization for prefix-closed systems

As pointed out in the previous section, the optimized version for prefix-closed languages takes into account that extensions of rejected strings are rejected. Before issuing a membership query to the teacher, we check whether we can deduce it from previous membership queries. In our setting, the optimized learner gives about the same execution time as the ordinary learner. Since we simulate the *Teacher* on the same computer, a query takes roughly the same

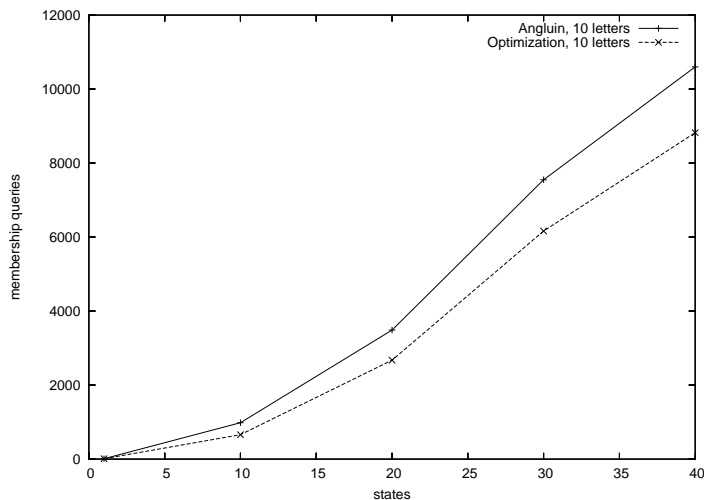


Fig. 5. Random prefix-closed examples learned with Angluin and optimization, number of letters fixed to 10.

amount of time as a table lookup. Note that in the setting where a concrete hardware system is learned, the time for a table lookup might be negligible compared to the time a membership query needs.

4.2.1 Random prefix-closed examples

The samples

On the optimized learner, we studied the same random prefix-closed examples as with the ordinary learner.

Experiences

We observe that the optimization yields noteworthy savings in terms of membership queries. To give an example, we have shown the number of membership queries with respect to the number of states in Figure 5 and Figure 6 for the number of letters fixed to 10 and 40, for the optimized version in comparison with Angluin’s version. We save in case of larger automata approximately 20% in both cases when using the optimization.

The particular example of size 100 states and 25 letters, from the previous subsection, took 12 hours, 96,000 membership queries, 29 equivalence queries and 160 MB of memory for our optimized learner.

The middle curve in Figure 3 shows membership queries with the optimized learner versus transitions for random prefix-closed examples. A very rough summary of our observations is $|membership\ queries| = k|\delta|^2$, with $k \approx 0.013$.

4.2.2 Real-world examples

The samples

We studied 6 transition systems of CCS processes. They are simple examples like buffers, vending machines, or several examples of schedulers and mutual

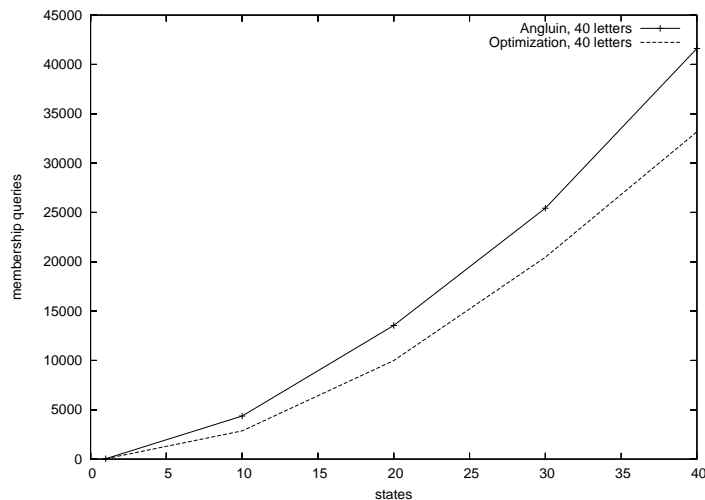


Fig. 6. Random prefix-closed examples learned with Angluin and optimization, number of letters fixed to 40.

exclusion protocols. Their number of states lie between 2 and 13 and the number of letters between 3 and 6. Note that we learned minimized DFA representations of the given protocols.

We failed to learn some larger protocols, namely some instances of parameterized schedulers, the Jobshop (77 states, 7 letters) and an ATM protocol (1715 states, 27 letters). The reason is that we did not invest effort into a good algorithm for finding counterexamples; it took too long to find counterexamples for the protocols in question. (Note that the execution time which we measure is independent of the time spent for finding counterexamples.) The ATM, though, failed due to lack of memory.

Experiences

The number of membership and equivalence queries, as well as execution time, are shown in Table 1.³

Comparing the number of membership queries of the optimized version with respect to Angluin’s algorithm, we saved about 60%. Details can be found in Table 2.

To check whether real-world examples show a different behavior in learning by the optimized algorithm, we compared them with random prefix-closed examples of the same sizes. Each result is an average over 6 to 8 fixed size random samples. The results are shown in Table 3.

We see that the optimized learner is often better on the protocols relative to random examples (see Table 4). On average the real-world examples required 7% fewer membership queries without and 35% with the optimization. This might indicate that real-world examples exhibit a certain structure which

³ In all tables *mq* is an abbreviation of the number of membership queries, *eq* of the number of equivalence queries and *opt* of the optimization for prefix-closed systems.

Protocol	states	letters	mq	eq	time (ms)	mq with opt	eq with opt	time with opt (ms)
Abp-lossy	3	3	22	2	65	9	2	1057
Buff3	9	3	202	5	2305	77	5	4907
Dekker-2	2	3	7	1	646	4	1	7
Peterson-2	2	3	7	1	352	4	1	288
Sched2	13	6	691	7	43031	115	7	48207
VMnew	11	4	513	7	26191	191	7	20091

Table 1
Learning real-world examples.

Protocol	mq	mq with opt	saved mq (%)
Abp-lossy	22	9	59
Buff3	202	77	62
Dekker-2	7	4	43
Peterson-2	7	4	43
Sched2	691	115	83
VMnew	513	191	63

Table 2
Saved membership queries with optimization.

states	letters	mq	eq	time (ms)	mq with opt	eq with opt (ms)	time with opt (ms)
3	3	22	2	153	12	2	115
9	3	233	5	1443	173	5	1384
2	3	7	1	25	4	1	10
13	6	992	8	10885	737	8	10989
11	4	497	7	5230	341	7	5408

Table 3
Random prefix-closed automata.

makes the algorithm perform better.

protocol	mq quo- tient	with opt
Abp-lossy	1	0.75
Buff3	0.87	0.45
Dekker-2	1	1
Peterson-2	1	1
Sched2	0.70	0.16
VMnew	1.03	0.56

Table 4

Random prefix-closed automata vs. real-world examples.

5 Conclusions and future work

Among the conclusions we draw from our experiences is the fact that random prefix-closed automata are harder to learn in comparison to completely randomly generated automata. For our random examples, the number of membership queries can roughly be described as linear in the number of transitions. Membership queries for our prefix-closed examples, in comparison, are approximately quadratic in transitions.

Moving deeper into the domain of prefix-closed automata we conclude that it is possible to reduce the number of membership queries by using an optimization specially shaped for these automata. The optimization reduces the number of membership queries considerably. For the randomly generated prefix-closed automata we measured a reduction of about 20%.

Turning our attention to the real-world examples we see that the optimization works much better, saving 60% membership queries relative to unoptimized learning. We also compared the result of learning real-world examples with randomly generated prefix-closed examples of the same size, in order to investigate if they behaved in the same manner. The result reveals a better performance for the real-world examples in terms of membership queries, especially with the optimization. This seems to imply that our real-world examples have a more suited structure for learning. Hopefully this observation can be used to optimize the learning process further.

Memory consumption is a problem we experienced when learning large models. In order to learn these models, we need more memory efficient data structures. Further optimizations for prefix-closed DFA are possible. For instance, one can save space and time by using the fact that there is exactly one non-final state.

References

- [1] Angluin, D., *Learning regular sets from queries and counterexamples*, Information and Computation **75** (1987), pp. 87–106.

- [2] Angluin, D., *Queries revisited*, in: N. Abe, R. Khardon and T. Zeugmann, editors, *Algorithmic Learning Theory, 12th International Conference 2001*, Lecture Notes in Computer Science **2225** (2001).
- [3] Corbett, J., M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby and H. Zheng, *Bandera : Extracting finite-state models from java source code*, in: *Proc. 22nd Int. Conf. on Software Engineering*, 2000.
- [4] Fernandez, J.-C., C. Jard, T. Jérón and C. Viho, *An experiment in automatic generation of test suites for protocols with verification technology*, Science of Computer Programming **29** (1997).
- [5] Grégoire, J.-C., G. J. Holzmann and D. A. Peled, editors, “The Spin Verification System,” DIMACS series **32**, American Mathematical Society, 1997, ISBN 0-8218-0680-7, 203p.
- [6] Groce, A., D. Peled and M. Yannakakis, *Adaptive model checking*, in: J.-P. Katoen and P. Stevens, editors, *Proc. TACAS '02, 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science **2280** (2002), pp. 357–370.
- [7] Hagerer, A., H. Hungar, O. Niese and B. Steffen, *Model generation by moderated regular extrapolation*, in: R.-D. Kutsche and H. Weber, editors, *Proc. FASE '02, 5th Int. Conf. on Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science **2306** (2002), pp. 80–95.
- [8] Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring and M. Trakhtenbrot, *STATEMATE: A working environment for the development of complex reactive systems*, IEEE Trans. on Software Engineering **16** (1990), pp. 403–414.
- [9] Holzmann, G., *Logic verification of ANSI-C code with SPIN*, in: K. Havelund, J. Penix and W. Visser, editors, *SPIN Model Checking and Software Verification: Proc. 7th Int. SPIN Workshop*, Lecture Notes in Computer Science **1885** (2000), pp. 131–147.
- [10] Hungar, H., O. Niese and B. Steffen, *Domain-specific optimization in automata learning*, in: *Proc. 15th Int. Conf. on Computer Aided Verification*, 2003, to appear.
- [11] Larsen, K., P. Pettersson and W. Yi, *UPPAAL in a nutshell*, Software Tools for Technology Transfer **1** (1997).
- [12] Schmitt, M., M. Ebner and J. Grabowski, *Test generation with autolink and testcomposer*, in: *Proc. 2nd Workshop of the SDL Forum Society on SDL and MSC - SAM'2000*, 2000.

Paper C

Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines with parameters. In *FASE*, volume 3922 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2006. Extended version. Reprinted with permission from Springer Verlag.

Regular Inference for State Machines with Parameters [★]

Therese Berg¹, Bengt Jonsson¹, Harald Raffelt²

¹ Department of Computer Systems, Uppsala University, Sweden
`{thereseb, bengt}@it.uu.se`

² Chair of Programming Systems and Compiler Construction, University of
Dortmund, Germany
`harald.raffelt@cs.uni-dortmund.de`

Abstract. Techniques for inferring a regular language, in the form of a finite automaton, from a sufficiently large sample of accepted and nonaccepted input words, have been employed to construct models of software and hardware systems, for use, e.g., in test case generation. We intend to adapt these techniques to construct state machine models of entities of communication protocols. The alphabet of such state machines can be very large, since a symbol typically consists of a protocol data unit type with a number of parameters, each of which can assume many values. In typical algorithms for regular inference, the number of needed input words grows with the size of the alphabet and the size of the minimal DFA accepting the language. We therefore modify such an algorithm (Angluin's algorithm) so that its complexity grows not with the size of the alphabet, but only with the size of a certain symbolic representation of the DFA. The main new idea is to infer, for each state, a partitioning of input symbols into equivalence classes, under the hypothesis that all input symbols in an equivalence class have the same effect on the state machine. Whenever such a hypothesis is disproved, equivalence classes are refined. We show that our modification retains the good properties of Angluin's original algorithm, but that its complexity grows with the size of our symbolic DFA representation rather than with the size of the alphabet. We have implemented the algorithm; experiments on synthesized examples are consistent with these complexity results.

1 Introduction

Model-based techniques for verification and validation of reactive systems, such as model checking and model-based test generation [6] have witnessed drastic advances in the last decades. They depend on the availability of a model, specifying the intended behavior of a system or component, which typically is developed during specification and design. However, in practice often no formal specification is available, or becomes

[★] Supported in part by the Swedish Research Council, and by the FP6 Network of Excellence ARTIST2

outdated as the system evolves over time. In, e.g., the telecommunication area, revision cycles are extremely short, and at the same time the short revision cycles necessitate extensive testing and verification. Therefore, there are many cases where the only means to attain correspondence between model and system component is to construct a model directly from the component. Such models can be constructed by static analysis techniques using its source code, as in software verification (e.g., [4, 8, 15, 16]). However, many system components, including peripheral hardware components, library modules, or third-party components do not allow static analysis of source code, implying that models must be constructed from observations of their external behavior.

The construction of models from observations of component behavior can be performed using techniques for regular inference. Such techniques have been used, e.g., to create models of environment constraints with respect to which a component should be verified, for regression testing to create a specification and a test suite [14, 17], to perform model checking without access to code or to formal models [13, 19], for program analysis [1], and for formal specification and verification [7]. For finite-state reactive systems, the regular inference problem means to infer a regular language (in the form of a deterministic finite automaton) from the answers to a finite set of *membership queries*, each of which asks whether a certain word is accepted by the system component under test (SUT) or not. There are several techniques (e.g., [2, 3, 9, 11, 18, 21, 23]) which use essentially the same basic principles. Given “enough” membership queries, the constructed automaton will be a correct model of the SUT. Angluin [2] and others introduce *equivalence queries* which check whether the regular inference procedure is completed; if not they are answered by a counterexample on which the current hypothesis and the SUT disagree.

We intend to use regular inference to construct models of communication protocol entities. Such entities typically communicate by messages that consists of a protocol data unit (PDU) type with a number of parameters, each of which can assume several values. The alphabet of such models is thus typically very large. Since existing algorithms for regular inference use a number of queries, which grows polynomially with the size of the alphabet, they are not well suited for this situation. If some PDU parameters are irrelevant or almost never used, the algorithm should not be disturbed by their presence.

In this paper, we modify an algorithm for inferring a regular language, so that it is better adapted for inferring system components with large alphabets that are built from a small set of action types, each of which

has a number of parameters. Most of these algorithms are based on similar principles: we choose Angluin’s algorithm [2] since it is well known, and since we have an existing implementation for this algorithm [20]. The problem of inferring state machines where messages have arbitrary parameters appears to be very challenging. As a first step, we will in this paper assume that all parameters are booleans, and that a SUT can be modeled as an automaton, in which each transition is labeled by a PDU type and a guard over its parameters. We assume that guards are conjunctions over positive and negated parameter values. Furthermore, we will not consider the problem of inferring parameters of possible output data, but only how input parameters affect the state changes of a state machine. Ideas for how to extend these rather restrictive limitations are sketched in the last section of the paper.

Algorithms for regular inference must represent the inferred automaton in terms of externally observable elements. A state is represented by a set $[u]$ of input words u such that the automaton after reading u reaches this state. For each input symbol a , the transition from $[u]$ for input a is constructed by determining which state is reached after reading ua . In the parameterized case, input symbols are of the form $\alpha(d_1, \dots, d_n)$, where α is an action type and d_1, \dots, d_n is a tuple of boolean parameter values. We could naively use Angluin’s algorithm to find the state reached after each of these 2^n different input symbols. Instead, we will strive to save work by assuming that from each automaton state, many of the input symbols have the same effect on the SUT, and can be regarded as equivalent. We can then construct a symbolic automaton representation, where the effect of each set of equivalent input symbols is represented by a transition from this state, labeled by a guard, i.e., a boolean expression over the parameters, which characterizes the equivalence class. In cases where the number of equivalence classes is small, we would like to perform the inference with less work (as measured by the number of membership queries) than by a naive application of Angluin’s algorithm.

Our inference algorithm maintains, for each inferred state, a partitioning of subsequent input symbols into assumed equivalence classes. Each class is represented by a small set of representative input symbols that (as far as we have observed) have the same effect on the SUT. If later, new information is obtained which contradicts this assumption, the equivalence class is split, thus also splitting transitions and generating more refined guards. The guard that labels a transition is obtained by a search procedure to identify precisely the effect of parameter values, inspired by work on learning of conjunctions, e.g., [18, Ch. 1.3].

In order to develop a consistent algorithm to do the above, we present in this paper two significant extensions of Angluin’s algorithm:

1. We generalize Angluin’s algorithm so that it can infer a “partially defined” automaton, which from each state defines the effect of a set of representative input symbols. The representative symbols are in general only a subset of all input symbols.
2. We define a mechanism for inferring guards of a parameterized system from the symbols in an underlying partially defined automaton, by replacing the representative symbols by guards that characterize the transitions represented by each symbol. Extra queries may need to be performed to determine guards more precisely.

Our resulting inference algorithm is intended to infer parameterized systems where guards of transitions use only a small subset of all parameters of a particular action type. We establish an upper bound on the number of posed membership queries, which is exponential in the number of parameters that appear in guards. In contrast, using Angluin’s original algorithm requires a number of membership queries which is exponential in the total number of parameters of input symbols. On the other hand, the number of equivalence queries may grow in our case, since we add possibilities to construct hypothesized automata based on less information than in the original algorithm. We have performed a set of experiments on synthesized examples, which confirm this picture.

Organization. The paper is organized as follows. In the next section, we review Angluin’s algorithm for inferring regular sets, and present a modification which can cope with the situation that the queries investigate different sets of suffixes for different prefixes. In Section 3, we present parameterized systems, and the technique to learn “partially defined” automata, from which guards of transitions are inferred. We prove that our algorithm retains good properties of Angluin’s original algorithm, and establish upper bounds on the number of performed queries. Section 4 describes how we have implemented the ideas of the preceding section, and Section 5 presents the outcome of experiments on synthesized examples. Conclusions are presented in Section 6.

2 Inference of Finite Automata

In this section, we review the ideas underlying Angluin’s algorithm, and present our generalization.

Let Σ be a finite alphabet of symbols. A *deterministic finite automaton (DFA)* over Σ is a structure $\mathcal{M} = (Q, \delta, q_0, F)$ where Q is a non-empty finite set of *states*, $q_0 \in Q$ is the *initial state*, $\delta : Q \times \Sigma \rightarrow Q$ is the *transition function*, and $F \subseteq Q$ is the set of *accepting states*. The transition function is extended from input symbols to words of input symbols in the standard way, by defining

$$\begin{aligned}\delta(q, \varepsilon) &= q \\ \delta(q, ua) &= \delta(\delta(q, u), a)\end{aligned}$$

An input word u is *accepted* iff $\delta(q_0, u) \in F$. The *language* accepted by \mathcal{M} , denoted by $\mathcal{L}(\mathcal{M})$, is the set of accepted input words.

Angluin's algorithm is designed to infer a (minimized) DFA \mathcal{M} from a set of queries, each of which reveals whether a certain word is accepted or not. The algorithm is formulated in a setting, where a so called *Learner*, who initially knows nothing about \mathcal{M} , is trying to infer \mathcal{M} by asking queries, which are of two kinds.

- A *membership query* consists in asking whether a word $w \in \Sigma^*$ is in $\mathcal{L}(\mathcal{M})$.
- An *equivalence query* consists in asking whether a hypothesized DFA \mathcal{H} is correct, i.e., whether $\mathcal{L}(\mathcal{H}) = \mathcal{L}(\mathcal{M})$. The *Oracle* will answer *yes* if \mathcal{H} is correct, or else supply a *counterexample*, which is a word u that is either in $\mathcal{L}(\mathcal{M}) \setminus \mathcal{L}(\mathcal{H})$ or in $\mathcal{L}(\mathcal{H}) \setminus \mathcal{L}(\mathcal{M})$.

The typical behavior of a *Learner* is to start by asking a sequence of membership queries, and gradually build a hypothesized DFA \mathcal{H} using the obtained answers. When the *Learner* feels that she has built a “stable” hypothesis \mathcal{H} , she makes an equivalence query to find out whether \mathcal{H} is equivalent to \mathcal{M} . If the result is successful, the *Learner* has succeeded, otherwise she uses the returned counterexample to revise \mathcal{H} and perform subsequent membership queries until converging at a new hypothesized DFA, etc.

Let us represent the information gained by the *Learner* at any point during the learning process, as a partial mapping Obs from Σ^* to $\{+, -\}$, where $+$ stands for *accepted* and $-$ for *rejected*. The domain $Dom(Obs)$ of Obs is the set of words for which membership queries have been performed, or which the *Oracle* has given as counterexamples in equivalence queries. An inference algorithm should prescribe how to transform Obs into a DFA $\mathcal{H} = (Q, \delta, q_0, F)$, which is *conformant* with Obs , in the sense that any word $u \in Dom(Obs)$ is accepted by \mathcal{H} if $Obs(u) = +$ and rejected by \mathcal{H} if $Obs(u) = -$. In general, there are many such automata, and the

problem to find a smallest (in number of states) such automaton is NP-complete [12]. Angluin and others circumvent this problem by prescribing conditions on $Dom(Obs)$, under which it is “easy” to find a unique smallest automaton. These conditions regard each word in $Dom(Obs)$ as the concatenation of a prefix and a suffix. The idea is that prefixes are candidates for representing states of the hypothesized automaton, whereas suffixes are used to distinguish the states.

Angluin [2] supports this prefix-suffix view by representing Obs in terms of an *observation table* \mathcal{T} , which is a partial function from a prefix-closed set $Dom(\mathcal{T}) \subseteq \Sigma^*$ of prefixes. For each $u \in Dom(\mathcal{T})$, $\mathcal{T}(u)$ is a partial function from a set $Dom(\mathcal{T}(u)) \subseteq \Sigma^*$ of suffixes to $\{+, -\}$. It is required that $\varepsilon \in Dom(\mathcal{T}(u))$ for each $u \in Dom(\mathcal{T})$. We write $Entries(\mathcal{T})$ to denote $\{(u, v) : u \in Dom(\mathcal{T}) \text{ and } v \in Dom(\mathcal{T}(u))\}$. An observation table \mathcal{T} represents the partial mapping Obs if $uv \in Dom(Obs)$ and $Obs(uv) = \mathcal{T}(u)(v)$ whenever $(u, v) \in Entries(\mathcal{T})$.

Define the *short prefixes* of an observation table \mathcal{T} , denoted $Sp(\mathcal{T})$, to be the set of words $u \in Dom(\mathcal{T})$ such that $ua \in Dom(\mathcal{T})$ for some $a \in \Sigma$. An observation table \mathcal{T} is *complete* if $ua \in Dom(\mathcal{T})$ for all $u \in Sp(\mathcal{T})$ and $a \in \Sigma$; it is *suffix-closed* if $(u, av) \in Entries(\mathcal{T})$ where $u \in Sp(\mathcal{T})$ and $a \in \Sigma$ implies that $(ua, v) \in Entries(\mathcal{T})$. For $u, u' \in Dom(\mathcal{T})$, let $u \approx_{\mathcal{T}} u'$ denote that $\mathcal{T}(u)(v) = \mathcal{T}(u')(v)$ whenever $v \in (Dom(\mathcal{T}(u)) \cap Dom(\mathcal{T}(u')))$. The table \mathcal{T} *partitioned* if $\approx_{\mathcal{T}}$ is an equivalence relation on $Dom(\mathcal{T}(u))$. A partitioned table is *closed* if whenever $(u, v) \in Entries(\mathcal{T})$ there is a $u' \in Sp(\mathcal{T})$ with $u \approx_{\mathcal{T}} u'$ and $v \in Dom(\mathcal{T}(u'))$; it is *consistent* if $ua \approx_{\mathcal{T}} u'a$ whenever $ua, u'a \in Dom(\mathcal{T})$ and $u \approx_{\mathcal{T}} u'$.

Angluin showed how to construct a unique minimal automaton from a complete, closed, and consistent observation table in the case that $Dom(\mathcal{T}(u))$ is the same for all $u \in Dom(\mathcal{T})$. Our goal in this section is to generalize this construction to the case where the set $Dom(\mathcal{T}(u))$ of suffixes may differ significantly for different prefixes $u \in Dom(\mathcal{T})$.

Definition 1. *Let \mathcal{T} be a partitioned, complete, closed, and consistent observation table. Define the DFA $\mathcal{T}/\approx_{\mathcal{T}}$ as (Q, δ, q_0, F) , where*

- $Q = Dom(\mathcal{T})/\approx_{\mathcal{T}}$, i.e., Q is the set of equivalence classes of $\approx_{\mathcal{T}}$,
- $\delta([u], a) = [ua]$ for $u \in Sp(\mathcal{T})$,
- $q_0 = [\varepsilon]$,
- $F = \{[u] : \mathcal{T}(u)(\varepsilon) = +\}$ □

Note how closedness and completeness ensures that we can define a transition for each equivalence class and symbol in Σ , and how consistency ensures that such transitions have a unique target equivalence class.

We are now ready to state a general theorem that gives constraints on any FSM that is conformant with an observation function.

Theorem 1 (Characterization Theorem). *Let \mathcal{T} be a partitioned, complete, closed, and consistent observation table which represents Obs . If \mathcal{T} is suffix-closed, then the DFA $\mathcal{T}/\approx_{\mathcal{T}}$ is the minimal automaton conformant with Obs .*

Angluin’s algorithm uses a specialization of the conditions in Theorem 1, where $Dom(\mathcal{T}(u))$ is the same for all $u \in Dom(\mathcal{T})$.

3 Inference of Parameterized Systems

In this section, we consider how to adapt the techniques of the previous section to a setting where symbols in the alphabet are messages with parameters, e.g., as in a typical communication protocol. Since the problem of inferring state machines where messages have arbitrary parameters appears to be very challenging, we will here assume that all parameters are booleans, and that a SUT can be modeled as an automaton, in which each transition is labeled by a PDU type and a guard over its parameters. We assume that guards are conjunctions over positive and negated parameter values. Furthermore, we will not consider the problem of inferring parameters of possible output data, but only how input parameters affect the state changes of a state machine.

Let Act be a finite set of *actions*, each of which has a nonnegative arity. Let Σ_{Act} be the set of *symbols* of form $\alpha(d_1, \dots, d_n)$, where α is an action of arity n , and d_1, \dots, d_n are booleans. We will use 0 and 1 to denote the boolean values *false* and *true*, respectively.

We assume a set of *formal parameters*, ranged over by p, p_1, p_2, \dots . A *parameterized action* is a term of form $\alpha(p_1, \dots, p_n)$, where α is an action α of arity n , and p_1, \dots, p_n are formal parameters. A *guard* for $\alpha(p_1, \dots, p_n)$ is a conjunction whose conjuncts are of form p_i or $\neg p_i$ with $p_i \in \{p_1, \dots, p_n\}$. We write \bar{p} for p_1, \dots, p_n and \bar{d} for d_1, \dots, d_n . A *guarded action* is a pair $(\alpha(\bar{p}), g)$, where $\alpha(\bar{p})$ is a parameterized action, and g is a *guard* for $\alpha(\bar{p})$. A guarded action $(\alpha(\bar{p}), g)$ denotes the set $\llbracket (\alpha(\bar{p}), g) \rrbracket = \{\alpha(\bar{d}) : g[\bar{d}/\bar{p}]\}$ of symbols, whose parameters satisfy g .

Definition 2 (Parameterized system). *Let Act be a finite set of actions. A parameterized system over Act is a tuple $\mathcal{P} = (Q, \longrightarrow, q_0, F)$, where*

- Q is a finite set of states,

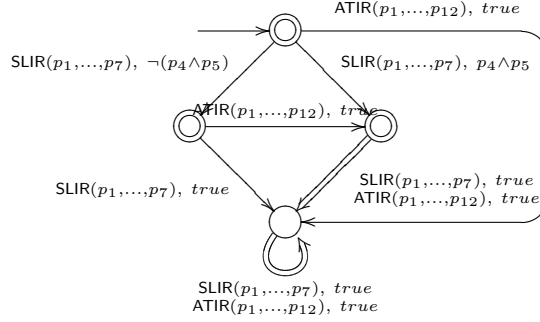


Fig. 1. Example of a parameterized system

- \longrightarrow is a finite set of transitions. Each transition is a tuple $\langle q, \alpha(\bar{p}), g, q' \rangle$, where $q, q' \in Q$ are states, and $(\alpha(\bar{p}), g)$ is a guarded action,
- $q_0 \in Q$ is the initial state, and
- $F \subseteq Q$ is a set of accepting states,

which is completely specified and deterministic, i.e., for each state q and symbol $\alpha(\bar{d})$, there is exactly one transition $\langle q, \alpha(\bar{p}), g, q' \rangle$ from q such that $\alpha(\bar{d}) \in \llbracket (\alpha(\bar{p}), g) \rrbracket$. \square

We write $q \xrightarrow{\alpha(\bar{p}), g} q'$ to denote that $\langle q, \alpha(\bar{p}), g, q' \rangle \in \longrightarrow$. A parameterized system is *expanded* if whenever $q \xrightarrow{\alpha(\bar{p}), g'} q'$ and $q \xrightarrow{\alpha(\bar{p}), g''} q''$, and in addition p_i or $\neg p_i$ is a conjunct of g' , then either p_i or $\neg p_i$ must be a conjunct of g'' . In other words, a parameterized system is expanded if all transitions from a state for some action test the same set of parameters. In Fig. 1 a fragment of a protocol provided by Mobile Arts AB [5] is given as an example of a parameterized system.

A parameterized system $\mathcal{P} = (Q, \longrightarrow, q_0, F)$ over Act denotes the DFA $\mathcal{M}_{\mathcal{P}} = (Q, \delta, q_0, F)$ over Σ_{Act} , where δ is defined by

$$\delta(q, \alpha(\bar{d})) = q' \quad \text{whenever} \quad q \xrightarrow{\alpha(\bar{p}), g} q' \text{ and } \alpha(\bar{d}) \in \llbracket (\alpha(\bar{p}), g) \rrbracket.$$

Note that δ is well-defined, since \mathcal{P} is completely specified and deterministic.

We will adapt Angluin's algorithm to inference of parameterized systems, in a situation where each symbol typically has many parameters, but for which the number of outgoing transitions from each state is small compared to the number of symbols in Σ_{Act} . Ideally, the effort needed to learn a parameterized system \mathcal{P} should be in proportion to the size of its

description as a parameterized system, and not to its number of states and $|\Sigma_{Act}|$, as is the case for Angluin’s algorithm.

To accomplish this, we make two extensions to Angluin’s algorithm. First, we must abandon the requirement that the constructed observation table \mathcal{T} be complete, since then $Dom(\mathcal{T})$ is at least $|\Sigma_{Act}|$ times larger than the number of states of the constructed automaton. Instead of requiring that \mathcal{T} be complete, $Dom(\mathcal{T})$ will for each $u \in Sp(\mathcal{T})$ contain a set of representative continuations $u\alpha(\bar{d})$, where $\alpha(\bar{d})$ is taken from a *subset* of Σ_{Act} which in general depends on u . The ambition is that for each transition of the SUT, labeled $\alpha(\bar{p}), g$, from the state represented by u , the table contains at least one continuation $u\alpha(\bar{d})$ for a representative symbol $\alpha(\bar{d})$ with $\alpha(\bar{d}) \in \llbracket(\alpha(\bar{p}), g)\rrbracket$.

Second, in order to construct a parameterized system from an incomplete observation table, we present a technique to construct guards from representative symbols. This implies asking additional queries in order to determine guards as precisely as possible. Of course, we do not know *a priori* how many transitions leave a particular state, or how the guards partition symbols into equivalence classes. Therefore we start with a coarse default partitioning into equivalence classes, which is refined “by need”. Whenever two words in the same equivalence class generate different reactions by the SUT, we split the equivalence class by introducing more guards.

In order to maintain a current hypothesis about guards, we augment the observation table \mathcal{T} by a *labeling function* γ , which to each prefix $ua \in Dom(\mathcal{T})$ assigns a guarded action $\gamma_u(a)$. The idea is that the constructed parameterized system, after having processed the input word u , will process the input symbol a using a transition labeled by $\gamma_u(a)$. We make the natural requirements that $a \in \llbracket\gamma_u(a)\rrbracket$, and that the labeling function should suggest guards that make the resulting automaton completely specified and deterministic, i.e., for each $u \in Sp(\mathcal{T})$, we have

- $\bigcup_{ua \in Dom(\mathcal{T})} \llbracket\gamma_u(a)\rrbracket = \Sigma_{Act}$, and
- $ua, ua' \in Dom(\mathcal{T})$ implies either $\llbracket\gamma_u(a)\rrbracket = \llbracket\gamma_u(a')\rrbracket$ or $\llbracket\gamma_u(a)\rrbracket \cap \llbracket\gamma_u(a')\rrbracket = \emptyset$.

The addition of a labeling function makes it natural to strengthen the notion of consistency, to allow a unique parameterized system to be constructed from an observation table with a labeling function.

Definition 3. A labeling function γ for an observation table \mathcal{T} is guard-consistent if for any $ua, u'a' \in \text{Dom}(\mathcal{T})$ such that $u \approx_{\mathcal{T}} u'$ and $\llbracket \gamma_u(a) \rrbracket \cap \llbracket \gamma_{u'}(a') \rrbracket \neq \emptyset$, we have $ua \approx_{\mathcal{T}} u'a'$.

Intuitively, whereas consistency states that extensions ua and $u'a$ in $\text{Dom}(\mathcal{T})$ of equivalent prefixes u and u' with the *same symbol* a should also be equivalent, guard-consistency requires that two symbols a, a' , whose labeling functions *overlap* should have equivalent extensions in $\text{Dom}(\mathcal{T})$. Note that guard-consistency as a special case implies that $ua \approx_{\mathcal{T}} u'a'$ whenever $ua, u'a' \in \text{Dom}(\mathcal{T})$ and $\llbracket \gamma_u(a) \rrbracket = \llbracket \gamma_{u'}(a') \rrbracket$.

We now have defined enough concepts to be able to define how to construct a parameterized system from an observation table with a labeling function.

Definition 4. Let Act be a finite set of actions. Let \mathcal{T} be a partitioned, closed, and consistent observation table, and let γ be a guard-consistent labeling function for \mathcal{T} . Define the parameterized system $\langle \mathcal{T}, \gamma \rangle / \approx_{\mathcal{T}}$ as $(Q, \longrightarrow, q_0, F)$, where

- $Q = \text{Dom}(\mathcal{T}) / \approx_{\mathcal{T}}$,
- $[u] \xrightarrow{\alpha(\bar{p}), g} [ua]$ whenever $ua \in \text{Dom}(\mathcal{T})$ and $\gamma_u(a) = (\alpha(\bar{p}), g)$, and u is the principal prefix in $[u]$,
- $q_0 = [\varepsilon]$, and
- $F = \{[u] : \mathcal{T}(u)(\varepsilon) = +\}$.

where for each equivalence class $[u]$ we have designated a unique principal prefix $u' \in [u]$ with $u' \in \text{Sp}(\mathcal{T})$. \square

Note that guard-consistency guarantees that different choices of principal prefixes result in equivalent parameterized systems.

In general, there are many different guard-consistent labeling functions for a given observation table. We therefore define an additional criterion which constrains how conjuncts may occur in guards of a labeling function. In a table \mathcal{T} , define a *witnessing pair* for a prefix $u \in \text{Sp}(\mathcal{T})$, action α , and index i , to be a pair of prefixes $u\alpha(\bar{d}), u\alpha(\bar{d}') \in \text{Dom}(\mathcal{T})$ such that

- $u\alpha(\bar{d}) \not\approx_{\mathcal{T}} u\alpha(\bar{d}')$, and
- $\bar{d} = (d_1, \dots, d_i, \dots, d_n)$ and $\bar{d}' = (d_1, \dots, d'_i, \dots, d_n)$ differ only in the i th parameter.

Definition 5. A labeling function γ for \mathcal{T} is well-witnessed if whenever $\gamma_u(a) = (\alpha(\bar{p}), g)$ then

- whenever p_i or $\neg p_i$ is a conjunct in g , then \mathcal{T} contains a witnessing pair for u , α , and i .
- there is a conjunct p_j or $\neg p_j$ of g such that \mathcal{T} contains a witnessing pair $u\alpha(\bar{d}), u\alpha(\bar{d}')$ for u , α , and j , such that $\alpha(\bar{d}) \in \llbracket(\alpha(\bar{p}), g)\rrbracket$. \square

Intuitively, the first requirement states that each conjunct of a guard g should be motivated by a witnessing pair in \mathcal{T} , which however need not contain a prefix that satisfies g . The second requirement states that g should be satisfied by the last symbol of at least one prefix in a witnessing pair.

We are now ready to state a theorem which relates a parameterized system $\langle \mathcal{T}, \gamma \rangle / \approx_{\mathcal{T}}$ constructed from an observation table \mathcal{T} , and the internal structure of the SUT.

We first adapt Theorem 1 to be sure that $\langle \mathcal{T}, \gamma \rangle / \approx_{\mathcal{T}}$ agrees with the observations.

Theorem 2. *Let \mathcal{T} be a partitioned, closed, and consistent observation table, and let γ be an $\approx_{\mathcal{T}}$ -compatible and guard-consistent labeling function for \mathcal{T} . If \mathcal{T} is suffix-closed, then the parameterized system $\langle \mathcal{T}, \gamma \rangle / \approx_{\mathcal{T}}$ is conformant with \mathcal{T} .*

Proof. The theorem follows by adapting Theorem 1 to incomplete observation tables, and the requirement that $a \in \llbracket \gamma_u(a) \rrbracket$ for all $ua \in \text{Dom}(\mathcal{T})$. \square

A more informative theorem, which can be seen as an analogue of Theorem 1 in [2], is as follows.

Theorem 3. *Let \mathcal{T} be a partitioned, closed, consistent, and suffix-closed observation table, and let γ be a guard-consistent and well-witnessed labeling function for \mathcal{T} . Let $\langle \mathcal{T}, \gamma \rangle / \approx_{\mathcal{T}}$ be $(Q, \longrightarrow, q_0, F)$ with n states. Let $\mathcal{P} = (R, \longrightarrow', r_0, G)$ be any expanded parameterized system which is conformant with \mathcal{T} . Then \mathcal{P} has at least n states and there is a partial surjective mapping h from R to Q such that*

- $h(r_0) = q_0$,
- $r \in G$ iff $h(r) \in F$,
- if \mathcal{P} has exactly n states then, whenever $h(r) = q$ and $q \xrightarrow{\alpha(\bar{p}), g} q'$, there are g', r' such that $r \xrightarrow{\alpha(\bar{p}), g'} r'$ with $h(r') = q'$ and $g' \implies g$.

This implies that if \mathcal{P} has n states then $\langle \mathcal{T}, \gamma \rangle / \approx_{\mathcal{T}}$ has at most as many transitions as \mathcal{P} .

Proof. Let us first define h . Define $h(r) = [u]$ for the word $u \in Sp(\mathcal{T})$ such that r agrees with u on all suffixes in $Dom(\mathcal{T}(u))$.

We first prove that h is well-defined. Suppose that a state $r \in R$ agrees with u and u' , both in $Sp(\mathcal{T})$, on all suffixes in $Dom(\mathcal{T}(u))$ and $Dom(\mathcal{T}(u'))$ respectively, and $[u] \neq [u']$. According to the definition of h then $h(r) = [u]$ and $h(r) = [u']$. But this is contradictory since \mathcal{T} is partitioned and then there exists a $v \in (Dom(\mathcal{T}(u)) \cap Dom(\mathcal{T}(u')))$ for which $\mathcal{T}(u)(v) \neq \mathcal{T}(u')(v)$ and therefore r cannot agree with both u and u' .

We prove that h is surjective. For each $[u]$, $u \in Sp(\mathcal{T})$, there is at least one such $r \in R$, namely $\delta_{\mathcal{P}}(r_0, u)$, where $\delta_{\mathcal{P}}$ is the transition function in $\mathcal{M}_{\mathcal{P}}$. Hence there are at least n states in \mathcal{P} .

Suppose \mathcal{P} has exactly n states. Then since h is surjective and $\langle \mathcal{T}, \gamma \rangle / \approx_{\mathcal{T}}$ has n states, h is injective. since otherwise there exist two different prefixes $u, u' \in Sp(\mathcal{T})$ who in \mathcal{P} lead to two different states $r, r' \in R$ and in $\langle \mathcal{T}, \gamma \rangle / \approx_{\mathcal{T}}$ to one state $[u] = [u'] = q \in Q$. But this is not possible since then it exists a $q' \in Q$ which is not in the range of h and this contradicts the fact that h is surjective.

The conditions on the initial states hold since $h(r_0) = \delta_{\mathcal{P}}(r_0, \varepsilon) = [\varepsilon] = q_0$.

As a result of the assumption $r \in Dom(h)$ and the definition of h there is a $u \in Sp(\mathcal{T})$ and $h(r) = [u]$. Then $r \in G$ iff $\mathcal{T}(u)(\varepsilon) = +$ since \mathcal{T} is conformant with \mathcal{P} . $\mathcal{T}(u)(\varepsilon) = +$ iff $[u] \in F$ according to Definition 4, hence $[u] = h(r) \in F$.

Assume now that $h(r) = [u] = q$, and $q \xrightarrow{\alpha(\bar{p}), g} q'$. There is an $a \in \llbracket \alpha(\bar{p}), g \rrbracket$ with $\gamma_u(a) = (\alpha(\bar{p}), g)$. Let $\langle r, \alpha(\bar{p}), g', r' \rangle$ be the transition from r such that $a \in \llbracket \alpha(\bar{p}), g' \rrbracket$. We then have $q' = [ua] = h(\delta_{\mathcal{P}}(r_0, ua)) = h(\delta_{\mathcal{P}}(\delta_{\mathcal{P}}(r_0, u), a)) = h(\delta_{\mathcal{P}}(r, a)) = h(r')$.

Finally lets show that $g' \implies g$. Assume that p_i is a conjunct in g . Then since \mathcal{T} is well-witnessed there exist $u\alpha(\bar{d}), u\alpha(\bar{d}') \in Dom(\mathcal{T})$ who only differ in place i in \bar{d} and \bar{d}' , and $u\alpha(\bar{d}) \not\approx_{\mathcal{T}} u\alpha(\bar{d}')$. Conjunct p_i or $\neg p_i$ is in g' otherwise $\delta_{\mathcal{P}}(r_0, u\alpha(\bar{d})) = \delta_{\mathcal{P}}(r_0, u\alpha(\bar{d}'))$ which would contradict that \mathcal{P} is conformant with \mathcal{T} . Since \mathcal{P} is expanded, p_i or $\neg p_i$ is in every guard for each transition from r and since $a \in \llbracket \alpha(\bar{p}), g' \rrbracket$ then p_i is a conjunct in g' . \square

Optimization The process of obtaining a well-witnessed labeling function may need a number of additional queries, which cause $Dom(\mathcal{T})$ to be extended. The requirement that $\approx_{\mathcal{T}}$ be an equivalence relation on $Dom(\mathcal{T})$ may then necessitate even more queries, which are not necessary

for making γ well-witnessed. To allow to save queries, we allow prefixes in $Dom(\mathcal{T})$ to be classified as either *essential* or *auxiliary*. We now say that an observation table is *partitioned* if

- For each $ua \in Dom(\mathcal{T})$, there is an essential $ua' \in Dom(\mathcal{T})$ with $\gamma_u(a') = \gamma_u(a)$,
- ε is an essential prefix, and
- $\approx_{\mathcal{T}}$ is an equivalence relation on essential prefixes in $Dom(\mathcal{T})$.

4 An Algorithm for Inference of Parameterized Systems

In this section, we present an algorithm for inferring parameterized systems, based on the concepts introduced in Section 3.

The basic idea of our algorithm is to perform membership queries until we have a suffix-closed, partitioned, closed, and consistent observation table with a guard-consistent and well-witnessed labeling function. We can then construct a conjecture and pose an equivalence query. As long as the table does not satisfy some condition mentioned in Theorem 3, this is handled as follows.

- If \mathcal{T} is not suffix-closed, i.e., there is a $(u, av) \in Entries(\mathcal{T})$ where $u \in Sp(\mathcal{T})$, such that $(ua, v) \notin Entries(\mathcal{T})$, then add (ua, v) to $Entries(\mathcal{T})$ (letting $\mathcal{T}(ua)(v) = \mathcal{T}(u)(av)$).
- If \mathcal{T} is not partitioned, i.e., $\approx_{\mathcal{T}}$ is not an equivalence relation, then there are $u, u', u'' \in Dom(\mathcal{T})$ such that $u \approx_{\mathcal{T}} u'$, $u \approx_{\mathcal{T}} u''$ but $\mathcal{T}(u')(v) \neq \mathcal{T}(u'')(v)$ for some v . In this case, ask a membership query for uv , whose result is entered as $\mathcal{T}(u)(v)$ to determine whether u should be equivalent to u' or u'' .
- If \mathcal{T} is not closed, then for some $ua \in Dom(\mathcal{T})$ we have $ua \not\approx_{\mathcal{T}} u'$ for all $u' \in Sp(\mathcal{T})$. We then add ua to $Sp(\mathcal{T})$ by adding, for each $\alpha \in Act$, some word of form $ua\alpha(\bar{d})$ to $Dom(\mathcal{T})$, and let $\gamma_{ua}(\alpha(\bar{d})) = (\alpha, true)$. Priority given to parameters \bar{d}' for which $\alpha(\bar{d}')v \in Dom(\mathcal{T}(ua))$ for some v , since suffix-closedness then requires that $ua\alpha(\bar{d}') \in Dom(\mathcal{T})$.
- If \mathcal{T} is not consistent, then we have two entries (ua, v) and $(u'a, v)$ in $Entries(\mathcal{T})$ with $\mathcal{T}(ua)(v) \neq \mathcal{T}(u'a)(v)$ but $u \approx_{\mathcal{T}} u'$. Then add (u, av) and (u', av) to $Entries(\mathcal{T})$ and enter the results from $\mathcal{T}(ua)(v)$ and $\mathcal{T}(u'a)(v)$, respectively.

The table must also be equipped with a labeling function γ , which is maintained during the algorithm. Initially, for each $u \in Sp(\mathcal{T})$ and each action α , we choose some values \bar{d} for the parameters of α , and

let $u\alpha(\bar{d}) \in \text{Dom}(\mathcal{T})$ with $\gamma_u(\alpha(\bar{d})) = (\alpha, \text{true})$. Whenever we add a prefix $u\alpha(\bar{d})$ to $\text{Dom}(\mathcal{T})$ the labeling function is updated in one of two ways. If there is not yet a prefix $u\alpha(\bar{d}') \in \text{Dom}(\mathcal{T})$ for any \bar{d}' we let $\gamma_u(\alpha(\bar{d})) = (\alpha, \text{true})$, otherwise we let $\gamma_u(\alpha(\bar{d})) = \gamma_u(\alpha(\bar{d}'))$, where $\alpha(\bar{d}')$ is the existing symbol such that $\alpha(\bar{d}) \in \llbracket \gamma_u(\alpha(\bar{d}')) \rrbracket$.

If $\text{Dom}(\mathcal{T})$ contains only one prefix $u\alpha(\bar{d})$ for each u and α , then γ is well-witnessed. However, if another prefix $u\alpha(\bar{d}')$ is entered, for which $u\alpha(\bar{d}) \not\approx_{\mathcal{T}} u\alpha(\bar{d}')$, this destroys the guard-consistency. We then have to refine the labeling function γ , and possibly also the partitioning into equivalence classes.

If γ is not guard-consistent, this may be because there are u , a , and a' such that $\gamma_u(a) = \gamma_u(a')$ but $ua \not\approx_{\mathcal{T}} ua'$. Let $\gamma_u(a)$ be $(\alpha(\bar{p}), g)$. In this case, we must split the guard g so that a and a' are assigned disjoint guards. In order to find an appropriate parameter for the splitting, and to keep γ well-witnessed, we find (e.g., by binary search) two tuples, $\bar{d} = (d_1, \dots, 1, \dots, d_n)$ and $\bar{d}' = (d_1, \dots, 0, \dots, d_n)$, of parameter values of α , with $\alpha(\bar{d}), \alpha(\bar{d}') \in \llbracket \gamma_u(a) \rrbracket$, which differ only in some parameter (with index, say, i), such that $\mathcal{T}(u\alpha(\bar{d}))(v) \neq \mathcal{T}(u\alpha(\bar{d}'))(v)$ for some v . We then add $(u\alpha(\bar{d}), v)$ and $(u\alpha(\bar{d}'), v)$ to $\text{Entries}(\mathcal{T})$, and update the labeling function so that all $ua'' \in \llbracket \gamma_u(a) \rrbracket$ now labeled by the guard $g \wedge p_i$ or $g \wedge \neg p_i$.

A second source of guard-inconsistency is that we can have two equivalent prefixes in $\text{Sp}(\mathcal{T})$ which have different partitionings of the next symbols, induced by the labeling function. It must then always be the case that there exist u, u', a , and a' such that $ua, u'a' \in \text{Dom}(\mathcal{T})$, $u \approx_{\mathcal{T}} u'$, and $a' \in \llbracket \gamma_u(a) \rrbracket$ but $\mathcal{T}(ua)(v) \neq \mathcal{T}(u'a')(v)$ for some v . A membership query for $ua'v$ should clarify the situation, either giving rise to a guard-inconsistency, or causing $u \not\approx_{\mathcal{T}} u'$ (and continuing processing it as an inconsistency).

When we have a partitioned, suffix-closed, consistent, and closed table with a well-witnessed and guard-consistent labeling function, we can construct a conjecture as described in Definition 4. The conjecture is provided to the oracle in an equivalence query and the oracle in turn either gives an affirmative answer or a counter-example. In the first case, the algorithm terminates and outputs the correct model. In the second case, the oracle returns a counter-example, i.e., a word u such that $\text{Obs}(u) = +$ but the provided automaton does not accept u (or vice versa). As in the standard algorithm of Angluin, we enter all prefixes of u into $\text{Dom}(\mathcal{T})$.

This will subsequently cause either an inconsistency and hence a "new" state, or a guard-inconsistency and hence a "new" transition.

Algorithm Query complexity We estimate the complexity of our algorithm in terms of a minimal expanded parameterized system which accepts exactly the language as the SUT. Let n be its number of states, let m be the number of transitions, let $|Act|$ be the number of actions in Act , let $|\Sigma_{Act}|$ be the number of symbols in Σ_{Act} , and let c be the length of the longest counter-example received from the oracle.

The expected bottle-neck in practice for an inference algorithm is the number of membership and equivalence queries, since queries often involve comparatively slow communication with an external device. Let us first estimate the number of equivalence queries. An equivalence query can either give rise to

- an inconsistency which results in a new state; this can occur at most n times, or
- a guard-inconsistency which results in splitting a guard; this can occur at most $m - n|Act|$ times.

Hence the algorithm performs at most $n + m - n|Act|$ equivalence queries.

Let us then estimate the number of membership queries. The number of membership queries required are dependent on the number of prefixes in $Dom(\mathcal{T})$ and the maximum number of suffixes in any $Dom(\mathcal{T}(u))$. Each $Dom(\mathcal{T}(u))$ contains at most n suffixes, since each time we add a new suffix to $Dom(\mathcal{T}(u))$ we separate at least a pair of prefixes into different equivalence classes. The number of prefixes in $Dom(\mathcal{T})$ is at most

- one for each equivalence class; totally n , plus
- one for each state and action, plus an extra essential pair of prefixes as witness for each transition, in total $n|Act| + 2m$, plus
- prefixes of counterexamples, in total $c(n + m - n|Act|)$.

Hence the number of membership queries performed by the algorithm is $O(cmn)$ (since $n|Act| \leq m$). We can contrast this with a naive application of Angluin's algorithm, which in the worst case requires $O(cn^2|\Sigma_{Act}|)$ membership queries. Thus, whereas a naive use of Angluin's algorithm uses a number of membership queries which grows linearly with $|\Sigma_{Act}|$, i.e., exponentially in the arity of actions, our algorithm grows exponentially only with the number of parameters of an action that is used in guards of transitions. It should be remarked that Angluin's treatment of counterexamples is poorly optimized, resulting in the factor c in the worst-case bound. Rivest and Shapire [21] have presented techniques for replacing the factor c by $\log c$, which should apply also to our algorithm.

5 Experimental Results

We are interested in examining how the performance of the inference algorithm for parameterized systems depends on the number of parameters that occur in guards in the transitions of the system and how it compares with a naive application of Angluin’s algorithm. Let us first define a measure for this. Let the *parameter complexity* for a state q and action α of a parameterized system, be the total number of different parameters used in guards on transitions from q labeled $\alpha(\bar{p})$. We want to investigate how the parameter complexity effects the number of membership and equivalence queries required by the algorithm. For this purpose, we have implemented our inference algorithm for parameterized systems. The implementation is in C++ as an extension of the LearnLib tool [20], developed at Dortmund University.

We measure performance on randomly generated parameterized systems and a small model of an instance of a protocol provided by Mobile Arts AB (see Fig. 1). The protocol was first modeled in LOTOS and then transformed into a DFA by the CAESAR/ALDEBARAN Development Package [10]. The protocol is a small fragment of the Network Presence Center (NPC) product of the company. The NPC is a middle-ware product to allow Mobile Network Operators to provide various presence information from the GSM network. The parameterized system model of the protocol has 4 states, one action with arity 12 and another with arity 7. The first action has on average parameter complexity 0 and the second 0.5. In the randomly generated systems we have used actions with arity 5, and generated automata in which each state-action pair has the same parameter complexity. We have varied the parameter complexity between 1 and 5. The systems has then been inferred both by our algorithm and by Angluin’s algorithm. The results of the experiments are summarized in Figure 2, where the left diagram shows the number of membership queries, and the right diagram shows the number of equivalence queries.

The left diagram shows that the number of membership queries for our algorithm grows exponentially with the parameter complexity of the system, whereas it is independent of parameter complexity for Angluin’s original algorithm. For a parameter complexity of less than 3, our algorithm performs better, but when parameter complexity increases, the overhead of our algorithm makes it clearly worse than Angluin’s. The right diagram shows that our algorithm always performs more equivalence queries than Angluin’s.

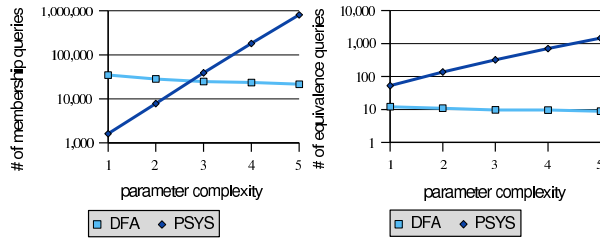


Fig. 2. Experimental results on random generated parameterized systems with 50 states and 5 parameters

Applying Angluin’s algorithm to Mobile Arts’ protocol fragment gives rise to 76000 membership queries and 3 equivalence queries, while our algorithm only requires 21 membership queries and 4 equivalence queries. The reason for this difference is the relatively low parameter complexity in the overall system in comparison to the high arity of its actions.

The higher number of equivalence queries for our algorithm is an expected consequence of the observation that our algorithm allows to construct equivalence queries that are based on less complete information than Angluin’s algorithm. In particular, we allow equivalence queries even if the refinement of equivalence classes of symbols is not completed. For higher parameter complexity (4 or 5), the difference in number of equivalence queries is significant. We believe that this explains the sharp growth of membership queries for parameter complexities 4 and 5, since a large number of equivalence queries gives rise to an explosion in membership queries that are caused by prefixes of counterexamples.

6 Conclusions

In this paper, we have adapted techniques for inference of finite automata from sets of observations, in order that they perform better for state machines whose symbols are generated from a small set of actions, each of which has a set of parameters. Our algorithm tries to find representative observations, from which we infer guards of transitions by techniques for inferring boolean expressions. Thus, our work indicates a way to combine techniques for inferring properties of data types with regular inference techniques for inferring reactive behavior. Our algorithm requires less observations in the case that only a subset of parameters are used to determine the behavior of the machine at each transition. Future work

includes to improve the handling of counterexamples in our tool, and to evaluate our techniques on a realistic communication protocol module.

Our framework limits us to handling inputs but not outputs. We therefore suggest possible solutions to include these. One approach is to infer a Mealy machine like Steffen et al. [22] but with our framework of handling parameterized input actions. The other approach is to use our framework but encode the input and output into parameterized actions of a parameterized system. This will of course blow up the alphabet, but the dependences between input and output will be recorded in boolean formulas which may lead to very compact models.

Acknowledgement At last we would like to thank Bernhard Steffen for helpful hints and discussions.

References

1. G. Ammons, R. Bodik, and J. Larus. Mining specificatoins. In *Proc. 29th ACM Symp. on Principles of Programming Languages*, pages 4–16, 2002.
2. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
3. J.L. Balcázar, J. Daz, and R. Gavaldá. Algorithms for learning finite automata from queries: A unified view. In *Advances in Algorithms, Languages, and Complexity*, pages 53–72. Kluwer, 1997.
4. T. Ball and S.K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. 29th ACM Symp. on Principles of Programming Languages*, pages 1–3, 2002.
5. J. Blom and B. Jonsson. Automated test generation for industrial erlang applications. In *Proc. 2003 ACM SIGPLAN workshop on Erlang*, pages 8–14, Uppsala, Sweden, Aug. 2003.
6. M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer Verlag, 2004.
7. J.M. Cobleigh, D. Giannakopoulou, and C.S. Pasareanu. Learning assumptions for compositional verification. In *Proc. TACAS '03, 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer Verlag, 2003.
8. J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proc. 22nd Int. Conf. on Software Engineering*, June 2000.
9. P. Dupont. Incremental regular inference. In L. Miclet and Colin de la Higuera, editors, *ICGI*, volume 1147 of *Lecture Notes in Computer Science*, pages 222–237. Springer, 1996.
10. H. Garavel, F. Lang, and R. Mateescu. An overview of cadp 2001, 2002. Newsletter.
11. E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.
12. E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

13. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In J.-P. Katoen and P. Stevens, editors, *Proc. TACAS '02, 8th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 357–370. Springer Verlag, 2002.
14. A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. In R.-D. Kutsche and H. Weber, editors, *Proc. FASE '02, 5th Int. Conf. on Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 80–95. Springer Verlag, 2002.
15. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. 29th ACM Symp. on Principles of Programming Languages*, pages 58–70, 2002.
16. G.J. Holzmann. Logic verification of ANSI-C code with SPIN. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification: Proc. 7th Int. SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 131–147, Stanford, CA, 2000. Springer Verlag.
17. H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *Proc. 15th Int. Conf. on Computer Aided Verification*, 2003.
18. M.J. Kearns and U.V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.
19. D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In J. Wu, S. T. Chanson, and Q. Gao, editors, *Formal Methods for Protocol Engineering and Distributed Systems, FORTE/PSTV*, pages 225–240, Beijing, China, 1999. Kluwer.
20. H. Raffelt, B. Steffen, and T. Berg. Learnlib: a library for automata learning and experimentation. In *FMICS '05: Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 62–71, New York, NY, USA, 2005. ACM Press.
21. R.L. Rivest and R.E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103:299–347, 1993.
22. Bernhard Steffen, Tiziana Margaria, Harald Raffelt, and Oliver Niese. Efficient test-based model generation of legacy systems. In *HLDVT'04: Proc. of the 9th IEEE Int. Workshop on High Level Design Validation and Test*, pages 95–100, Sonoma (CA), USA, November 2004. IEEE Computer Society Press.
23. B.A. Trakhtenbrot and J.M. Barzdin. *Finite automata: behaviour and synthesis*. North-Holland, 1973.