Domain-Specific Optimization in Automata Learning

Hardi Hungar, Oliver Niese, and Bernhard Steffen

University of Dortmund
{Hardi.Hungar, Oliver.Niese, Bernhard.Steffen}@udo.edu

Abstract. Automatically generated models may provide the key towards controlling the evolution of complex systems, form the basis for test generation and may be applied as monitors for running applications. However, the practicality of automata learning is currently largely preempted by its extremely high complexity and unrealistic frame conditions. By optimizing a standard learning method according to domain-specific structural properties, we are able to generate abstract models for complex reactive systems. The experiments conducted using an industry-level test environment on a recent version of a telephone switch illustrate the drastic effect of our optimizations on the learning efficiency. From a conceptual point of view, the developments can be seen as an instance of optimizing general learning procedures by capitalizing on specific application profiles.

1 Introduction

1.1 Motivation

The aim of our work is improving quality control for reactive systems as can be found e.g. in complex telecommunication solutions. A key factor for effective quality control is the availability of a specification of the intended behavior of a system or system component. In current practice, however, only rarely precise and reliable documentation of a system's behavior is produced during its development. Revisions and last minute changes invalidate design sketches, and while systems are updated in the maintenance cycle, often their implementation documentation is not. It is our experience that in the telecommunication area, revision cycle times are extremely short, making the maintenance of specifications unrealistic, and at the same time the short revision cycles necessitate extensive testing effort. All this could be dramatically improved if it were possible to generate and then maintain appropriate reference models steering the testing effort and helping to evaluate the test results. In [6] it has been proposed to generate the models from previous system versions, by using learning techniques, and incorporating further knowledge in various ways. We call this general approach *moderated regular extrapolation*, which is tailored for a posteriori model construction and model updating during the system's life cycle. The general method includes many different theories and techniques [14].

W.A. Hunt, Jr. and F. Somenzi (Eds.): CAV 2003, LNCS 2725, pp. 315–327, 2003.

In this paper we address the major bottlenecks of moderated regular extrapolation (cf. Sec. 1.3) – the *size* of the extrapolated models and the *performance* of the learning procedure – by application-specific optimization: Looking at the structural properties of the considered class of telecommunication systems allowed a significant advancement. This approach provides us with a key towards successfully exploiting automata learning, a currently mostly theoretical research area, in an industrial setting for the testing of telecommunication systems. Moreover it illustrates again that practical solutions can be achieved by 'intelligent' application-specific adaptation of general purpose solutions. In fact, we observed in our experiments that, even after eliminating all the 'low hanging fruits' resulting from prefix closedness and input determinism, the symmetry and partial order reductions still added a performance gain of almost an order of magnitude (cf. Sec. 3). As the impact of this optimization is non-linear, this factor will typically grow with the size of the extrapolated models.

Learning and testing are two wide areas of very active research — but with a rather small intersection so far, in particular wrt. practical application. At the theoretical level, a notable exception is the work of [13,4]. There, the problem of learning and refining system models is studied, e.g. with the goal of testing a given system for a specific property, or correcting a preliminary or invalidated model. In contrast, our paper focuses on enhancing the practicality of learning models to be able to capture real-life systems. This requires powerful optimizations of the concepts presented in [6,14].

1.2 Major Applications

Regression testing provides a particularly fruitful application scenario for using extrapolated models. Here, previous versions of a system are taken as the reference for the validation of future releases. By and large, new versions should provide everything the previous version did. I.e., if we compare the new with the old, there should not be too many essential differences. Thus, a model of the previous system version could serve as a useful reference to much of the expected system behavior, in particular if the model is abstract enough to focus on the essential functional aspects and not on details like, for instance, exact but irrelevant timing issues. Such a reference could be used for:

- Enhanced test result evaluation: Usually, success of a test is measured only via very few criteria. A good system model provides a much more thorough evaluation criterion for success of a test run.
- Improved error diagnosis: Related to the usage above, in case of a test error, a model might expose already very early some discrepancy to expected behavior, so that using a model will improve the ability to pinpoint an error.
- Test-suite evaluation and test generation: As soon as a model is generated, all the standard methods for test generation and test evaluation become applicable (see [5,2] for surveys about test generation methods).

1.3 Learning Finite Automata — Theory and Practice

Constructing an acceptor for an unknown regular language can be difficult. If the result is to be exact, and the only source of information are tests for membership in the considered language and a bound on the number of states of the minimal accepting deterministic finite automaton (DFA) for the language, the worst-case complexity is exponential [10] in the number of states of the acceptor. Additional information enables learning in polynomial time: The algorithm named L* from [1] requires tests not only for membership, but also for equivalence of a tentative result with the language to be captured. The equivalence test enables the algorithm to always construct a correct acceptor in polynomial time. If the equivalence test is dropped, it is still possible to learn approximately in polynomial time, meaning that with high probability the learned automaton will accept a language very close to the given one [1].

We base our model construction procedure on L*. To make this work in practice, there are several problems to solve. First of all, the worlds of automata learning and testing, in particular the testing of telecommunication systems, have to be matched. A telecommunication system cannot be readily seen as a finite automaton. It is a reactive, real-time system which receives and produces signals from large value domains. To arrive at a finite automaton, one has to abstract from timing issues, tags, identifiers and other data fields. Different from other situations where abstraction is applied, for instance in validation by formal methods, here it is of crucial importance to be able to reverse the abstraction. Learning is only feasible if one can check actively whether a given abstract sequence corresponds to (is an abstraction of) a concrete system behavior, and L* relies heavily on being able to have such questions answered. I.e., in order to be able to resolve the *membership queries*, abstract sequences of symbols have to be retranslated into concrete stimuli sequences and fed to the system at hand. This problem has been solved with the help of a software enhancement of an existing, industrial test environment [11,12].

A major cause of the difficulties is the fact that L* – as all its companions – does not take peculiarities of the application domain into account. Though a suitable abstraction of a telecommunication system yields indeed a finite automaton, it is a very special one: Its set of finite traces forms a prefix-closed language, every other symbol in the trace is an output determined by the inputs received by the system before, and, assuming correct behavior, the trace set is closed under rearrangements of independent events. It was the last property, which caused most problems when adapting the learning scenario. In fact, identifying independence and the concrete corresponding exploitation for the reduction of memberships queries both turned out to be hard. However, since physical testing, even if automated as in our case, is very time intensive, these optimizations are vital for practicality reasons.

In the following, Sec. 2 prepares the ground for our empirical studies and optimizations, Sec. 3 elaborates on the application-specific characteristics of the considered scenario, and presents the implemented optimizations together with their effects. Finally Sec. 4 draws some conclusions and hints at future work.

2 Basic Scenario

In this section, we set the stage for our application-specific optimizations: Sec. 2.1 describes the considered application scenario, Sec. 2.3 our application-specific elaborations of the underlying basic learning algorithm together with the major required adaptations, and Sec. 2.4 describes our experimentation scenario, which allows us to evaluate our technique in an industrial setting.

2.1 Application Domain: Regression Testing of Complex, Reactive Systems

The considered application domain is the functional regression testing of complex, reactive systems. Such systems consist of several subcomponents, either hardware or software, communicating with and affecting each other. Typical examples for this kind of systems are *Computer Telephony Integrated (CTI) systems*, like complex *Call Center solutions*, embedded systems, or web-based applications. To actually perform tests, we use a tool called *Integrated Test Environment* (*ITE*) [11,12] which has been applied in research and in industrial practice for different tasks.

One important property for regression testing is that results are reproducible, i.e., that results are not subject to accidental changes. In the context of telecommunication systems, one observes that a system's reaction to a stimulus may consist of more than one output signal. Those outputs are produced with some delay, and in everyday application some of them may actually occur only after the next outside stimulus has been received by the system. Usually, functional testing does not care for exact timing. So the delays do not matter much. But it is very important to be able to match outputs correctly to the stimuli which provoked them. Thus, it is common practice to wait after each stimulus to collect all outputs. Most often, appropriate timeouts are applied to ensure that the system has produced all responses and settled in a "stable" state. If all tests are conducted in this way, for interpreting, storing and comparing test results, a telecommunication system can be viewed as an *I/O-automaton*: a device which reacts on inputs by producing outputs and possibly changing its internal state. It may also be assumed that the system is *input enabled*, i.e. that it accepts all inputs regardless of its internal state.

This is not all we need for our learning approach to work. First of all, we would like to look at the system as a propositional, that is finite, object. This means that we would like to have only finitely many components, input signals and output signals. W.r.t. the number of components, again we are helped by common testing practice. The complexity of the testing task necessitates a restriction to small configurations. So only a bounded number of addresses and component identifications will occur, and those can accordingly be represented by propositional symbols. Other components of signals can be abstracted away, even to the point where system responses to stimuli get deterministic – which is another important prerequisite for reproducible, unambiguously interpretable test results.

Summarizing, we adopted the following common assumptions and practices from real-life testing.

1. Distinction between stimuli and responses
2. Matching reactions to the stimuli that cause them
3. Restriction to finite installations
4. Abstraction to propositional signals

This leads to a view of a reactive/telecommunication system as a propositional, input-deterministic I/O-automaton.

Definition 1. *An* input/output automaton *is a structure* $\mathcal{S} = (\Sigma, A_I, A_O, \rightarrow, s_0)$*, consisting of a finite, non-empty set* Σ *of states, finite sets* A_I *and* A_O *of input, resp. output, actions, a transition relation* $\rightarrow \subseteq \Sigma \times A_I \times A_O^* \times \Sigma$*, and a unique start state* s_0*. It is called* input deterministic *if at each state* s *there is at most one transition for each input starting from that state. It is* input enabled *if there is at least one transition for each input.*

I/O automata according to this definition are not to be confused with the richer structures from [8]. As we are not concerned with automata algebra, we can use this simple form here[1]. As explained above, the system models we consider are all finite-state, input-deterministic I/O automata. Additionally, we can assume that they are input enabled.

2.2 L*: A Basic Learning Algorithm

Angluin describes in [1] a learning algorithm for determining an initially unknown regular set exactly and efficiently. To achieve this aim, besides the alphabet A of the language two additional sources of information are needed: a *Membership Oracle* (MO), and an *Equivalence Oracle* (EO). A *Membership Oracle* answers the question whether a sequence σ is in the unknown regular set or not with `true` or `false`, whereas an *Equivalence Oracle* is able to answer the question whether a conjecture (an acceptor for a regular language) is equivalent to the unknown set with `true` or a counterexample.

The basic idea behind Angluin's algorithm is to systematically explore the system's behavior using the membership oracle and trying to build the transition table of a deterministic finite automaton with a minimal number of states. During the exploration, the algorithm maintains a set S of state-access strings, a set of strings E distinguishing the states found so far, and a finite function T mapping strings of $(S \cup S \cdot A) \cdot E$ to $\{0, 1\}$, indicating results of membership tests. This function T is kept in the so-called *observation table* $\mathcal{OT}$ which is the central data structure of the algorithm and from which the conjectures are read off. For a detailed discussion about L* please refer to [1].

[1] Another common name for our type of machine is *Mealy Automaton*, only we permit a string of output symbols at each transition.

2.3 Application-Specific Adaptations to L*

Before the basic L* algorithm can be used in the considered scenario, some adaptations have to be made:

1. Representing I/O automata as ordinary automata, and
2. Practical implementation of a membership and an equivalence oracle

Formal Adaptations I/O automata differ from ordinary automata in that their edges are labelled with inputs and outputs instead of just one symbol. Obviously we could view a combination of an input and the corresponding output symbols as one single symbol. In our scenario this would result in a very large alphabet, in particular because outputs of an I/O automaton are sequences of elementary output symbols. Running L* with such a large alphabet would be very inefficient. So we chose to split an edge labelled by a sequence of one input and several output symbols into a sequence of auxiliary states, connected by edges with exactly one symbol. In this way, we keep the alphabet and the observation table small (even though the number of states increases). Further reductions, which are possible because of the specific structure of the resulting automata, prohibit a negative impact of the state increase on the learning behavior. They are discussed in Sec. 3.
Given a system represented by an input-deterministic, input-enabled I/O automaton $\mathcal{S}$, our adaptation of L* will learn a minimal DFA equivalent to $\mathcal{DFA}(\mathcal{S})$, the formally adapted version of $\mathcal{S}$. This equivalent DFA can be constructed rather straightforwardly: for each transition of $\mathcal{S}$, a set of corresponding transitions, with transient states in-between, will be created, where each transition is now labelled with a single alphabet symbol only. At each state (new or old), transitions for alphabet symbols not already labelling a transition exiting this states are introduced which end at an artificial error state.

Additionally the adequate realization of a membership and an equivalence oracle is required:

Membership Oracle Membership queries can be answered by testing the system we want to learn. This is not quite as simple as it sounds, mainly because the sequence to be tested is an abstract, propositional string, and the system on the other hand is a physical entity whose interface follows a real-time protocol for the exchange of digital (non-propositional) data. For processing a membership query a so-called test graph for the *ITE* has t be constructed, which describes the sequence of stimuli and expected responses, to actually being able to execute membership queries. In essence, the *ITE* bridges the gap between the abstract model and the concrete system.

Although the tests are automatically generated, membership queries are very expensive: The automatic execution of a single test took approximately 1.5 minutes during our experimentation because of timeouts of the system that have to be considered. Note that the long execution time is due to the generous timeouts that are specified for telecommunication systems. But as in almost every

real-time system timeouts are of central importance, it is expected, that the test execution time affects the overall performance of such approaches in general. Therefore it is very important to keep the number of such membership queries low.

Equivalence Oracle For a black-box system there is obviously no reliable equivalence check – one can never be sure whether the whole behavior spectrum of a system has been explored. But there are approximations which cover the majority of systems pretty well. The basic idea is to scan the system in the vicinity of the explored part for discrepancies to the expected behavior. One particularly good approximation is achieved by performing a systematic conformance test in the spirit of [3] which looks at source and goal of every single transition. Another possibility lies in checking consistency within a fixed lookahead from all states. In the limit, by increasing the lookahead, this will rule out any uncertainty, though at the price of exponentially increasing complexity.

In our experiments, it was sufficient to use the next system outputs as a lookahead for obtaining satisfactory results. We expect, however, that this will have to be extended for capturing more complex cases. We additionally foresee the integration of expert knowledge into the learning scenario, e.g. by checking temporal-logic constraints on tentative results (see also [6]).

2.4 Basic Learning in Practice

With our framework established so far, we are able to learn models for finite installations of (a part of) a complex call center solution (cf. [11], [12, Fig. 4]). A midrange telephone switch is connected to the public telephone network and acts as a 'normal' telephone switch to the phones. Additionally, it communicates directly via a LAN or indirectly via an application server with CTI applications that are executed on PC's.

Like the phones, CTI applications are active components: they may stimulate the switch (e.g. initiate calls), and they also react to stimuli sent by the switch (e.g. notify incoming calls). Moreover the applications can be seen as logical devices, which cannot only be used as a phone, but form a compound with physical devices, so that they can complement each other, e.g. an application can initiate a call on a physical phone based on a user's personal address book.

The technical realization of the necessary interface to this setup is provided by the *ITE*, in particular the *Test Coordinator*, which controls in particular two different test tools, i.e. a test tool to control the telephone switch (*Husim*) and one for the clients (*Rational Robot*). This is described in more detail in [12].

We have carried out our experiments on four finite installations, each consisting of the telephone switch connected to a number of telephones (called "physical devices"). The telephones were restricted differently in their behavior, ranging from simple on-hook (↑) and off-hook (↓) actions of the receiver to actually performing calls (→). The four installations are listed below. Note that we have two versions of the first three scenarios, depending on the observability of the signal (*hookswitch*).

$\underline{S_1}$: 1 physical device (A), $A_I = \{A\uparrow, A\downarrow\}$,
$A_O = \{initiated_A, cleared_A, [hookswitch_A]\}$.
$\underline{S_2}$: 2 physical devices (A, B), $A_I = \{A\uparrow, A\downarrow, B\uparrow, B\downarrow\}$,
$A_O = \{initiated_{\{A,B\}}, cleared_{\{A,B\}}, [hookswitch_{\{A,B\}}]\}$.
$\underline{S_3}$: 3 physical devices (A, B, C), $A_I = \{A\uparrow, A\downarrow, B\uparrow, B\downarrow, C\uparrow, C\downarrow, \}$,
$A_O = \{initiated_{\{A,B,C\}}, cleared_{\{A,B,C\}}, [hookswitch_{\{A,B,C\}}]\}$.
$\underline{S_4}$: 3 physical devices (A, B, C), $A_I = \{A\uparrow, A\downarrow, A\rightarrow B, B\uparrow, B\downarrow, C\uparrow, C\downarrow\}$,
$A_O = \{initiated_{\{A,B,C\}}, cleared_{\{A,B,C\}}, originated_{A\rightarrow B}, established_B\}$

The output events indicate which actions the telephone switch has performed on the particular input. For instance, $initiated_A$ indicates that the switch has noticed that the receiver of device A is off the hook, whereas $initiated_{\{A,B\}}$ denotes that we have both events $initiated_A$ and $initiated_B$. With $originated_{A\rightarrow B}$ the switch reports that it has forwarded the call request from A to B. In a more general setting, more system responses would lead to more transient states in the learned DFA.

The third column of Tab. 1 in Sec. 3.2 points out impressively that for learning even relatively small scenarios the number of membership queries is quite large. Therefore it is clearly unavoidable to reduce the number of membership queries drastically. In fact, we would have had difficulties to compute these figures in time for the publication without using some reductions. To solve this problem additional filters have been added to the connection from L* to the two oracles. These filters reduce the number of queries to the oracles by answering queries themselves in cases where the answer can be deduced from previous queries. In our experiments, we used properties like determinism, prefix closure and independence of events for filter construction. The next section formally develops this novel way of applying profile-dependent optimization to the learning process, which we consider a key for the practicality of automata learning.

3 Optimized Learning

3.1 Theory

This section describes several optimizations given through rules which can be used to filter the membership queries. Common to all rules is that they are able to answer the membership queries from the already accumulated knowledge about the system. They will be presented in the form `Condition` $\Rightarrow$ $\ldots\{$`true`, `false`$\}$. `Condition` refers to the $\mathcal{OT}$ in order to find out whether there is an entry in the table stating that some string σ is already known to be member of the set to be learned or not, i.e. whether we are interested in $T(\sigma)$.

The filter rules that will be presented are based on several properties of (special classes of) reactive systems. They are to some extent orthogonal to each other and could be applied independently. This potentially increases the usability both of our current concrete approach and of the general scenario. Given another learning problem, a possible way to adapt the learning procedure is to identify a profile of the new scenario in terms of specific properties and to optimize the learning process by means of the corresponding filter rules.

Prefix-Closure: In testing a reactive system, the observations made are finite prefixes of the potentially infinite sequences of inputs and outputs. Any error (i.e., non-acceptance of a string) implies that also no continuation of the string will be observed. Other than in general regular languages, there is no switching from non-accepting to accepting. Correspondingly, the DFAs resulting from translating an I/O automaton have just one sink as a non-accepting state. The language we want to learn is thus *prefix closed*. This property directly gives rise to the rule Filter 1, where each prefix of an entry in $\mathcal{OT}$, that has already been rated `true` (i.e. is a member of the target language), will be evaluated with `true` as well.

Filter 1 (Positive Prefix) $\exists \sigma' \in A^*.\ T(\sigma;\sigma') = 1 \Longrightarrow MO(\sigma) = \texttt{true}$

The contraposition states that once we have found a sequence σ that is rated `false`, all continuations have also to be rated `false`. This yields the rule Filter 2.

Filter 2 (Negative Prefix) $\exists \sigma' \in prefix(\sigma).\ T(\sigma') = 0 \Longrightarrow MO(\sigma)=\texttt{false}$

Together, these two rules capture all instances of membership queries which are avoidable when learning prefix-closed languages. Already these rather simple rules have an impressive impact on the number of membership queries as will be shown in Tab. 1.

Input Determinism Input determinism ensures that the system to be tested always produces the same outputs on any given sequence of inputs (cf. Definition 1). This implies that replacing just one output symbol in a word of an input-deterministic language cannot yield another word of this language.

Proposition 1. *Let $\mathcal{S}$ be an input-deterministic I/O automaton and let furthermore $\sigma \in \mathcal{L}(\mathcal{DFA}(\mathcal{S}))$. Then the following holds.*

1. *If there exists a decomposition of $\sigma = \sigma';x;\sigma''$ with $x \in A_O$, then $\forall y \in A \setminus \{x\}.\ \sigma';y;\sigma'' \notin \mathcal{L}(\mathcal{S})$.*
2. *If there exists a decomposition of $\sigma = \sigma';a;\sigma''$ with $a \in A_I$, then $\forall x \in A_O.\ \sigma';x;\sigma'' \notin \mathcal{L}(\mathcal{S})$.*

The first property says that each single output event is determined by the previous inputs. It may be emphasized that this property is of crucial importance for learning reactive systems, as a test environment has no direct control over the outputs of a system. If the outputs were not determined by the inputs, there would be no way to steer the learning procedure to exhaustively explore the system under consideration.

The second property reflects the fact that the number of output events in a given situation is determined, and that we wait with the next stimulus until the system has produced all its responses. This is useful but not as indispensable as the first property. The corresponding filter rules are straightforward:

Filter 3 (Input Determinism) $\exists x \in A_O, y \in A, \sigma', \sigma'' \in A^*.\sigma = \sigma'; x; \sigma'' \wedge T(\sigma'; y; \sigma'') = 1 \wedge x \neq y \implies MO(\sigma) = \texttt{false}$

Filter 4 (Output Completion) $\exists a \in A_I, x \in A_O.\sigma'; x \in prefix(\sigma) \wedge T(\sigma'; a) = 1 \implies MO(\sigma) = \texttt{false}$

Independence of Events: Reactive systems exhibit very often a high degree of parallelism. Moreover in telecommunication systems several different components of one type, here e.g. telephones, can often be regarded as generic so that they are interchangeable. This can lead to symmetries, e.g. a device A behaves like device B.

Partial order reduction methods for communicating processes [9,15] deal with this kind of phenomena. Normally these methods can help avoiding to examine all possible interleavings among processes. However, as we will illustrate here, these methods can also be used to avoid unnecessary membership queries using the following intuition:

Independence If device A can perform a certain action and afterwards device B can perform another, and those two actions are independent, then they can be performed in different order (i.e., device B starts) as well.

Symmetries If we have seen that device A can perform a certain action we know that the other (similar) devices can perform it as well.

This implies that the membership question for a sequence σ can be answered with `true` if an equivalent sequence σ' is already in $\mathcal{OT}$, where two sequences are equivalent if they are equal up to partial order and symmetries. Using $C_{po,sym}$ to denote the *partial order completion* with respect to *symmetries*, i.e. the set of all equivalent sequences up to the given partial order and symmetry, we obtain the following filter rule.

Filter 5 (Partial Order) $\exists \sigma' \in C_{po,sym}(\sigma).T(\sigma') = 1 \implies MO(\sigma) = true$

Whereas the general principle of Filter 5 is quite generic, the realization of $C_{po,sym}$ strongly depends on the concrete application domain. However, the required process may always follow the same pattern:

1. An expert specifies an application-specific **independence relation**, e.g. *Two independent calls can be shuffled in any order.*
2. During an *abstraction* step the concrete component identifiers will be replaced by generic place holders with respect to an appropriate symmetry.
3. σ is inspected if it contains independent subparts with respect to the independence relation.
4. All possible re-orderings will be computed.
5. The generic place holders will be *concretized* again (with all possible assignments due to symmetries).

Table 1. Number of Membership Queries

Scenario	States	no filter	Fil. 1 & 2	Factor	Fil. 3 & 4	Factor	Fil. 5	Factor	Tot. Factor
S_1	4	108	30	3.6	15	2.0	14	1.1	7.7
S'_1	8	672	181	3.7	31	5.8	30	1.0	22.4
S_2	12	2,431	593	4.1	218	2.7	97	2.2	25.1
S'_2	28	15,425	4,080	3.8	355	11.5	144	2.5	107.1
S_3	32	19,426	4,891	4.0	1,217	4.0	206	5.9	94.3
S'_3	80	132,340	36,374	3.6	2,007	18.1	288	7.0	459.5
S_4	78	132,300	29,307	4.5	3,851	7.6	1,606	2.4	81.1

In the following we will discuss how $C_{po,sym}$ works in the context of telephony systems by presenting a simple example. Consider a test run, where device A calls device B and after that device B answers the call so that it will be established. As a last action, device C also picks up the receiver. This action of C is independent from the previous ones, as device C is not involved in the connection between A and B. Therefore it does not matter at which time C performs its action.

In a first step the dependencies between the involved devices will be computed and this leads to two independent subsequences: one containing the call between device A and B, and the other contains the `hookoff` action by device C together with the responses of the switch to it. After that by an anonymization step every concrete device will be replaced by an *actor name* $(\alpha_1, \alpha_2, \alpha_3)$, i.e. every device will be treated as a generic device. This reflects the fact, that if we have observed once that e.g. device A can perform a certain action every other equivalent device can perform it as well. For the further processing it is of intrinsic importance that this abstraction step is reversible. In the next step, from these two independent subsequences, which describe a partial order, a directed acyclic graph representing all possible interleavings is computed. Finally, all concrete sequences that can occur through binding the actors again to the concrete devices (without using a device identifier twice) are determined, in order to e.g. treat the sequence where C is in connection with A and B picks up its receiver at some point as equivalent to the original one. More precisely we perform a concretization step where the abstract actions, referencing the different actors α_i, together with the set of available concrete devices in the considered setting $(Dev = \{A, B, \ldots\})$, will be transformed into concrete actions, suitable for the further processing within the learning algorithm.

3.2 Practice

In the following, we discuss the impact of the filters to the scenarios defined in Sec. 2.4. We have run the learning process with all available filters applied in a cumulative way, i.e., when using Filters 3 and 4, also Filters 1 and 2 were used. The results are summarized in Tab. 1. The "Factor" columns in the table provide the additional reduction factor in the number of membership queries achieved by successively adding new filters.

As one can see we were able to reduce the total number of membership queries in all scenarios drastically. In the most drastic case (S_3'), we only needed a fraction of a percent of the number of membership queries that the basic L* would have needed. In fact, learning the corresponding automaton without any optimizations would have taken about 4.5 months of computation time.

The prefix reduction (Filters 1 and 2) has a similar impact in all considered scenarios. This seems to indicate that it does not depend very much on the nature of the example and on its number of states.

The other two reductions (input determinism and partial order) vary much more in their effectiveness. Note that the saving factor increases with the number of states. Shifting attention to the number of outputs and the lengths of output sequences between inputs, these seem to have a particular high impact on the effects of the Filters 3 and 4. This can be seen by comparing the scenarios S_i with their counterparts S_i'. In these counterparts an additional output event is modelled, the hookswitch event, which occurs very frequently, namely after each of the permitted inputs.

One would expect that the impact of Filter 5, which covers the partial-order aspects, will increase with the level of independency. And indeed we find this conjecture confirmed by the experiments. S_1 has only one actor so that there is no independence, which results in a factor of 1. As the number of independent devices increases, the saving factor increases as well, see for instance the figures for S_2 and S_3. It is worth noting that the number of states does not seem to have any noticeable impact on the effectiveness of this filter, as the reduction factor more or less remains equal when switching from S_i and S_i'. Compared to S_3, the saving factor in S_4 decreases. This is due to the fact that an action has been added in S_4 that can establish dependencies between two devices, namely the initiation of a call.

4 Conclusion

We have demonstrated that with the right modifications to the learning procedure, and the right environment for observing system, system models may in fact be learned in practice. Modifying the learning algorithm by filtering out unnecessary queries enabled us to perform the experiments easily, and in general it provides a flexible approach which permits fast adaptations to different application domains. We are convinced that, at least for certain application scenarios, automata learning will turn out to be a powerful means for quality assurance and the control of system evolution.

For fixed settings, modifications to the algorithm itself should be made. For instance, it would not be hard to tailor L* to cope better with prefix-closed I/O-automata. However we conjecture that even after such a tailoring the presented filters will have a significant impact. A direct adaptation of the learning algorithm for capturing partial order and symmetry is far more involved. E.g., one is tempted to learn a reduced language whose partial-order closure is the full set of strings instead of the full language. However, it is known that the set

of strings reduced wrt. a partial order need not be regular [9]. Thus there is still enormous potential, both for further optimizations and for basic research.

5 Acknowledgement

We thank Tiziana Margaria for fruitful discussions, as well as the anonymous referees for many helpful comments on early versions of this paper.

References

1. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2(75):87–106, 1987.
2. E. Brinksma, and J. Tretmans. Testing transition systems: An annotated bibliography in *Proc. of MOVEP'2k*, pages 44–50, 2000.
3. T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.
4. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Proc. TACAS '02*, LNCS 2280, pages 357–370. Springer Verlag, 2002.
5. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proc. of the IEEE*, 84:1090–1126, 1996.
6. A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. In *Proc. of FASE '02*, LNCS 2306, pages 80–95. Springer Verlag, 2002.
7. M.J. Kearns and U.V. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, 1994.
8. N. Lynch and M. Tuttle. An introduction to input/ouput automata. *CWI Quaterly*, 2(3):219–246, 1989.
9. A. Mazurkiewicz. Trace theory. In *Petri Nets, Applications and Relationship to other Models of Concurrency*, LNCS 255, pages 279–324. Springer Verlag, 1987.
10. E.F. Moore. Gedanken-experiments on sequential machines. *Annals of Mathematics Studies (34), Automata Studies*, pages 129–153, 1956.
11. O. Niese, B. Steffen, T. Margaria, A. Hagerer, G. Brune, and H. Ide. Library-based design and consistency checks of system-level industrial test cases. In *Proc. of FASE '01*, LNCS 2029, pages 233–248. Springer Verlag, 2001.
12. O. Niese, T. Margaria, A. Hagerer, B. Steffen, G. Brune, H. Ide, and W. Goerigk. Automated regression testing of CTI-systems. In *Proc. IEEE ETW' 01*, pages 51–57, IEEE Press, 2001.
13. D. Peled, M.Y. Vardi, and M. Yannakakis. Black box checking. In *Proc. FORTE/PSTV '99*, pages 225–240. Kluwer Academic Publishers, 1999.
14. B. Steffen, and H. Hungar. Behavior-Based Model Construction. In *Proc. of VMCAI '02*, LNCS 2575, pages 5–19. Springer Verlag, 2002.
15. A. Valmari. On-the-fly verification with stubborn sets. In *Proc. of CAV '93*, LNCS 697, pages 397–408. Springer Verlag, 1993.