Int J Softw Tools Technol Transfer (2004) 6: 4–14 / Digital Object Identifier (DOI) 10.1007/s10009-004-0139-8

Behavior-based model construction

Hardi Hungar[*], **Bernhard Steffen**

Computer Science Department, University of Dortmund, Germany
e-mail: hungar@offis.de, steffen@cs.uni-dortmund.de

Published online: 1 July 2004 – © Springer-Verlag 2004

Abstract. In this paper, we review behavior-based model construction from a point of view characterized by *verification, model checking*, and *abstraction*. It turns out that *abstract interpretation* is the key to scaling known learning techniques for practical applications, that *model checking* may serve as a teaching aid in the learning process underlying the model construction, and that there are various synergies with other validation and *verification* techniques. We will illustrate our discussion by means of a realistic telecommunications scenario, where the underlying system has grown over the last two decades, the available system documentation consists of not much more than user manuals and protocol standards, and the revision cycle times are extremely short. In this situation, behavior-based model construction provides a sound basis, e.g., for test-suite design and maintenance, test organization, and test evaluation.

Keywords: Automata learning – Model checking – Abstraction – Testing

1 Introduction

Most systems in use today lack adequate specifications or make use of un(der)specified components. In fact, the much propagated component-based software design style naturally leads to underspecified systems, as most libraries only provide very partial specifications of their components. Moreover, typically, revisions and last-minute changes hardly enter the system specification. We observed this dilemma in the telecommunications area: the revision cycle times are extremely short, which makes the maintenance of specifications unrealistic, and at the same time the short revision cycles necessitate extensive testing efforts. More generally, the lack of documentation is felt in many places, of which quality control is one of the most conspicuous. Moderated regular extrapolation [6] has been proposed to overcome this situation: techniques known from automata learning have been adapted to cope with structures adequate for modeling reactive systems (like telecommunications systems) and to faithfully incorporate expert knowledge for guiding the learning process.

In this paper, we review the process of moderated regular extrapolation from a point of view characterized by *verification, model checking*, and *abstraction*. It turns out that *abstraction* is the key to adapting known learning techniques for our application scenario, that *model checking* is a good means for guiding the learning process, and that there are synergies with other *validation* and *verification* techniques that find their natural place: *testing* directly provides the observations the learning process is based upon, *assertion-based methods* allow for the capture of data and computation, and monitoring is at the same time an ideal application domain and a practical means for evaluating the quality of the learned models.

In this paper,[1] we will focus on the role of abstraction, which is at the heart of the whole learning procedure, while the aspects of model checking and verification will only be sketched.

Abstraction

Regular extrapolation is a process of iterated abstraction and refinement. While abstraction is necessary to achieve regular models (finite-state machines) at all, refinement is a means of obtaining a tailored compensation of too rigid abstraction steps, i.e., an aspect-specific enhancement of an abstraction. Technically, we can distinguish three kinds of abstraction, all of which naturally arise during the model construction process:

[*] *Present address:* OFFIS, R&D Division Safety-Critical Systems, Oldenburg, Germany

[1] This paper is an extended version of [10].

Complete abstractions. Given an observation level, many details may be hidden in the concrete system without affecting the learning process. In our application scenario, we focus on the protocol level that is the essential one (not only) for telecommunications systems. This allows us to completely ignore the message contents without losing any information at the protocol level, as the protocol behavior does not depend on the carried content. Thus skipping the content part provides in fact a *complete* abstract interpretation [4, 18]. The same is true of quantitative timing information, as long as the qualitative system behavior does not depend on time durations.

Safe approximations. Complete abstractions are insufficient for arriving at a regular model, as we typically deal with complex observational properties that are undecidable or at least not yet captured by current abstraction techniques. Thus we must deal with a loss of expressive power. Safe approximations, although typically imprecise, guarantee the preservation of properties in one direction: either properties proved at the abstract level are guaranteed to be true also at the concrete level (underapproximation) or vice versa (overapproximation). Whereas safety of approximations is essential for program analysis, in particular if the analysis results form the basis for program transformations, the approximations of our model construction approach are typically not safe. Rather, they lead to *faithful* hypotheses in the following sense:

Faithful hypotheses. Behavior-based model construction starts by collecting finite behavioral traces that are then *extrapolated* into regular models by means of automata learning techniques: automata learning builds model structures that distinguish states only on the basis of witnesses; states are identified until a distinguishing trace is found. As state identification increases the number of possible runs, learning leads to a behavioral *overapproximation* of the observed portion of the system. On the other hand, currently unobserved behavior is not modeled at all. Thus safety cannot be guaranteed in any direction. Rather, automata learning techniques iteratively construct minimal hypothesis automata consistent with all the observations. This is inherited by our regular extrapolation technique, which, additionally, comprises expert knowledge.

Our discussion will focus on:

- *Actor abstraction*, which abstracts individual objects to roles;
- The L^**-learning* algorithm (adapted for our purposes), which iteratively constructs the minimal hypothesis automaton; and
- *Partial-order reduction*, based on an externally given causality relation.

Model checking

We use temporal-logic formulas to specify the expert knowledge about the considered system; this includes constraints about the protocol, security policies, or functional requirements. These formulas can then be verified on each hypothesis model. Discrepancies leading to counterexamples are used to guide the learning process.

However, and perhaps even more importantly, regular extrapolation can and should be regarded as an aid for verification techniques like model checking to extend their applicability, similar to the approach of black-box checking in [17]: first, a behavioral model is constructed, then properties are checked on the model. Even legacy systems may then profit from model checking.

Verification/validation

Besides model checking, we employ various verification and validation techniques. Answering membership queries, the central activity of most automata learning techniques, essentially requires "classical" system tests, and constructed hypothesis models may well be used for monitoring the running system (online verification). Revealed discrepancies between the system and the model may then be analyzed to determine whether the system or the model should be modified. This analysis typically requires a manual effort, a fact that is reflected in our calling this approach *moderated* regular extrapolation.

In the following section, we present our application scenario before we characterize the role of model construction in Sect. 3. Then, Sect. 4 sketches "classical" automata learning, presents the model structure adequate for our application domain, and adapts the presented learning algorithm accordingly. Finally, Sects. 5 and 6 show how one can exploit the specific system structure for significantly speeding up the learning process, and Sect. 7 draws some conclusions and gives directions for future work.

2 Application scenario

We developed the methodology with the aim of applying it to complex reactive systems like those found in the telecommunications area. Such systems consist of several subcomponents, either hardware or software, communicating with and affecting each other. Typical examples of this kind of system are *Computer Telephony Integrated (CTI) systems*, such as complex *call-center solutions*, embedded systems, or Web-based applications. In the following two subsections, we present a concrete setting and the corresponding modeling.

2.1 A concrete setting

Our case study is based on the CTI system depicted in Fig. 1. Like most CTI systems, it employs an instance of the *Computer-Supported Telephony Applications (CSTA)* protocol for the communication between their components. A typical CSTA record contains several components. Some of them convey essential information relevant to modeling, others can be safely ignored. An example record may have the following components, described in the format `field`: value – meaning.

`invokeID:` 58391 – a number to identify the protocol frame.
`operation-value:` 21 – indicating the reporting of some external event (this is introduced by the observer functionality used by us to get information about internal actions in the telephone system).
`event-specific info:` hookswitch – name of the external event.

`device dialing number:` 500 – number of the device that issued the event.
`hookswitchOnHook:` TRUE – reports about the state of the external device
`timestamp:` 20001010095551

For most modeling purposes, it is sufficient to project this record (which, in fact, contains even more components in practice) onto something as abstract as `(hookswitchOnHook,500)`.

As an example system we take a telephone switch from a call-center installation as sketched in Fig. 1. The telephone switch is connected to the ISDN 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 PCs. 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).

2.2 Modeling

Systems as described above operate in real time in an environment exhibiting much parallelism. An interaction between components sometimes consists of several records sent back and forth. So when observing a CTI system in operation, one notices that sequences of records belonging to different interactions will often overlap.

We decided (for complexity reasons) not to model the real-time aspects. Additionally, just as is done in current test practice, we do not try to capture erroneous behavior in stress situations where interactions happen quickly and reactions to different stimuli may occur interleaved. Instead, we collect all the system's reactions to each single stimulus by waiting until the system has reached a stable state. Usually, this can be realized with timeouts. By neglecting timing issues in this way, we arrive at a view of reactive systems as input-deterministic I/O transition systems. Just like Mealy automata, I/O-transition systems are devices that react to inputs by producing outputs and possibly changing their internal state:

Definition 1.
An input/output transition system *is a structure* $\mathcal{S} = (\Sigma, A_I, A_O, \rightarrow, s_0)$, *consisting of*

- *A countable, nonempty set* Σ *of states;*
- *Countable 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, and it is called* input enabled *if there is at least one transition for each input at every state.*

Input determinism is a very important property when one tries to capture the behavior spectrum of a given system by systematically applying stimuli and observing the system's reactions. Another property indispensable for our approach is the finiteness of the model. There are two obstacles to viewing a given system as a finite entity:

1. Usually, the number of components connected to a system like the telephone switch might be rather

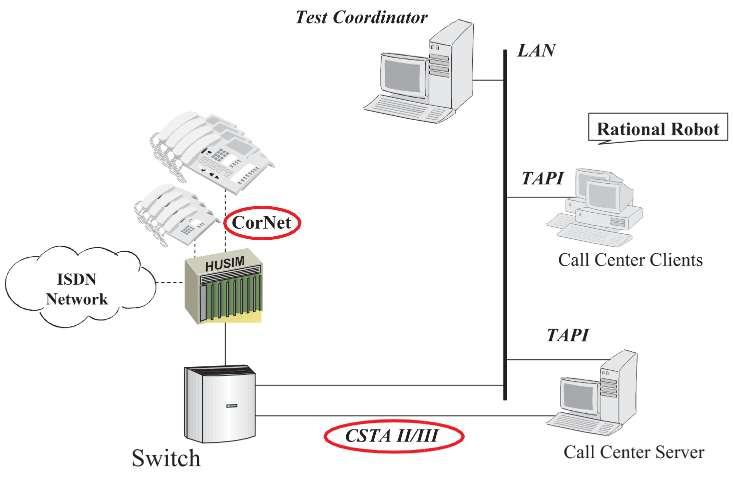

Fig. 1. Overview of the CTI system to be learned

large. Modeling a system with the maximal number of components (if known) would be impractical. Also, a new release might increase this parameter, thus invalidating the model. And, last but not least, such a large model would not reveal much additional information about the system. In fact, both protocol specifications and practical tests usually work with small, finite instantiations of a system environment. We do the same and thereby arrive at a manageable system size where address spaces can be represented by discrete symbols.[2]

2. Protocols may contain items like timestamps and tags. While we can easily abstract from timestamps when modeling at the functional level, tags in general might constitute a more severe problem. Tags are used, e.g., in protocols to reference some previous record (or incomplete exchange) unambiguously. Our answer to this problem is to restrict stimulation to use at most a bounded number of tags and to reuse tags whenever some exchange is completed. For this to work, it must be specified when exchanges (referenced by tags) terminate. In many cases, this can be done by either temporal-logic formulas or automata. Then, the restriction to finitely many tags is similar to the restriction to finite address spaces: it is natural and mirrors common practice. For our example application, tags appear only when CTI components enter the modeling focus. They are irrelevant if only telephones are connected to the switch.

3 Behavior-oriented model construction

Figure 2 sketches briefly our iterative approach. It starts with a model (initially empty) and a set of observations. The observations are gathered from a reference system in the form of traces. The notion *reference system* is due to the regression testing scenario: there, a previous system version is available and we want to generate a corresponding model as a means to check a new version for consistency with the previous one.

The observations can be obtained either *passively*, i.e., by observing a running reference system, or, better, *actively*, i.e., by stimulating a reference system through test cases. The set of traces (i.e., the observations) is then preprocessed, extrapolated, and used to extend the current model. After extension, the model is completed through several techniques, including adjustment to expert specifications. Finally, the current hypothesis for the model is validated, which can lead to new observations (in terms of counterexamples), and this closes the learning cycle.

[2] It may be argued that in some cases (e.g., protocol verification) it can even be proved to be safe to restrict attention to some small instantiation. These arguments, however, all require quite some elaborate side conditions that cannot be established in our real-life application domain, at least not without some very substantial work.

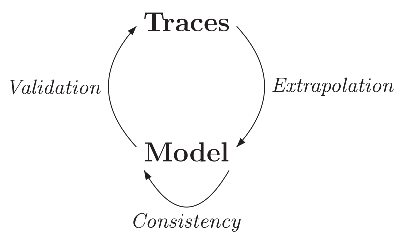

Fig. 2. Generation of models

The realization of this approach makes considerable use of automata operations and related techniques. The basis is given by standard automata operations like homomorphisms and boolean operations like *meet, join*, and *complement* [8].

These standard operations are complemented by some specific operations for extrapolation. Particularly important are abstraction mappings more general than language homomorphisms, together with a particular folding operation, which introduces cycles in the finite traces by identifying similar states. This borrows from automata learning techniques as discussed in [1] but requires substantial adaptation.

The adequate incorporation of expert knowledge requires further techniques. Here temporal logic [3] serves to formulate specifications that limit the model from above, i.e., experts formulate properties that they believe to be true of the system, and extrapolation results should be limited by them. Temporal-logic model checking is employed to check adherence of the model to these constraints. Counterexamples generated by the model checker in case of a violation are studied to pinpoint the source of the discrepancy.

Another way in which expert knowledge enters the method concerns specifying independence relations between events. By employing ideas of partial-order approaches [14, 20], this leads to a generalization from (randomly) sequential observations to parallel patterns.

Finally, the validation of models draws on testing theory (cf., e.g., [13] for a survey) to generate stimuli sequences that reveal wrongly identified states and missed behavior.

Summarizing, our method is composed of the following five ingredients:

1. *Testing,* for stimulating the system to be modeled in order to collect behavior sequences.
2. *Automata learning,* to guide the testing and to combine the results into an approximate model.
3. *Model checking,* to make use of expert knowledge for model validation.
4. *Manual interaction,* for conflict resolution.
5. *Change management,* to cope with revisions.

In the following two sections, we explain what has to be done to adapt automata learning to arrive at a practical model construction method for reactive systems, and we present some important optimizations.

4 Automata learning

After a brief description of the background of automata learning, this section presents the model structure adequate for our application domain, adapts the L* learning algorithm accordingly, and explains how the required membership and equivalence oracle can be implemented in our scenario.

4.1 Background

Let us assume that a regular language is given, so that we can test arbitrary strings for membership in the language, and that we want to construct a (minimal) acceptor.

The worst-case complexity of this learning problem is exponential in the (assumed to be known) number of states of the automaton to be learned [15]. The argument that proves the difficulty of exact learning relies on *combination-lock automata*. A combination lock within a finite automaton is a set of states linked by a sequence of transitions so that the last state can only be reached by exactly one sequence of symbols – any deviation will leave the combination lock and lead to other states. Whether an automaton to be learned contains such a lock can be detected only by testing all sequences of the length of the longest combination lock potentially contained in the automaton at hand, which results in an exponential number of tests.

In practice, however, such locks are not very common, and the probability of an arbitrary automaton containing one is low. So, approximate learning (PAC learning, [19]), which means that with high probability the learned automaton will accept a language very close to the given one, does not suffer from the combination-lock effect. And indeed it is known that approximate learning can be done in polynomial time if membership tests are available [1]. Experiments have provided evidence that this theoretical result is relevant for practice (see, for instance, [12]): random automata can be learned pretty well.

In our scenario, the system models to be learned are not random. One might even argue that some protocols are in some way similar to combination locks in that they impose sequences to be enacted in some prescribed order. Yet the combination-lock argument does not apply here:

- These sequences tend to be of bounded length, and
- They do not arise randomly: they can typically be found in manuals and protocol descriptions, as well as in existing test suites.

Whereas the first reason is only of theoretical interest, as the bounds are typically high enough to cause serious trouble, the latter reason is another evidence for the need for (and power of) moderation: the required knowledge can be loosely specified and used for guiding the learning process in an efficient way. So there is well-founded evidence that the combinatorial explosion can be controlled in our application scenario.

One way to perform the learning of automata is via the algorithm L* [1]. To factor out insecurities from approximation, L* relies on an *equivalence oracle* in addition to the more standard *membership oracle*. The equivalence oracle provides counterexamples if an automaton constructed from the information available so far does not match the given language. With this additional resource, finite automata can be learned exactly in polynomial time in the number of states of the original system. This is due to the immense power of the equivalence oracle, which, e.g., may instantaneously detect combination locks.

We use L* as the basis of our learning procedure. To make it work, it has to be adapted in several ways, as it solves learning in an idealized setting. Neither the membership oracle nor the equivalence oracle is available in practice, and they therefore have to be substituted by other means.

We solve membership queries by testing. In the presence of nontrivial abstractions, this is not an easy task. An important ingredient in solving this problem is a tool called *Integrated Test Environment (ITE)* [7, 16], which has been applied to a number of different tasks in research and in industrial practice. From a sequence of stimuli, the ITE generates a test program, using predefined code blocks for stimuli and additional glue code. Glue code and code blocks solve the problems connected with generating, checking, and identifying nonpropositional protocol elements like tags, timestamps, and identifiers. In essence, the ITE bridges the gap between the abstract model and the concrete system. The generated test program is interfaced to the system to be learned with the help of additional test hardware.

The equivalence oracle is more difficult to substitute. If no additional information about the system is available, there is of course no reliable equivalence check – one can never be sure whether the whole behavior spectrum of a system has been explored. But there are approximations of the equivalence oracle that cover the majority of systems pretty well. This is the basis of the theoretical result suggesting that PAC learning can be done in polynomial time with membership queries alone. The basic idea is to scan the system in the vicinity of the explored part looking for discrepancies with respect to the expected behavior.

A particularly good approximation is achieved by performing a test in the spirit of [2, 21]. In essence, this test checks each transition in the hypothesis/conjecture automaton by validating that the target state behaves properly along a sequence that distinguishes it from all other states of the conjecture automaton. This test has polynomial complexity – we have additional means at our disposal in the form of formalized expert knowledge. This is described in more detail in Sect. 4.5.

In addition to oracle substitutions, we have to adapt the algorithm itself from its original application domain – deterministic acceptors – to input-deterministic input/output transition systems in order to capture our application domain.

*4.2 Learning finite automata with L**

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. The required information is organized in the central data structure of the algorithm called *Observation Table* ($\mathcal{OT}$), which comprises the results of the membership queries and represents the intermediate conjectures.

Definition 2 (Observation Table). *Given an alphabet A, a finite, prefix-closed set $S \subset A^*$, and a finite set $E \subset A^*$, an $\mathcal{OT}$ is a two-dimensional array with rows for each string in $S \cup S \cdot A$ and columns for strings in E and entries in* $\{0, 1\}$*. An entry $\mathcal{OT}(s, e) = 1$ is interpreted in the context of the algorithm L* that the sequence $s \cdot e$ is a member of the regular set, while* 0 *means that it is not.*

1. *An $\mathcal{OT}$ is called **closed** if*

$$\forall t \in S \cdot A.\, \exists s \in S.\, row(t) = row(s).$$

2. *An $\mathcal{OT}$ is called **consistent** if*

$$\forall s_1, s_2 \in S.\, row(s_1) = row(s_2)$$
$$\Rightarrow\ \forall a \in A.\, row(s_1; a) = row(s_2; a).$$

The algorithm starts with the initial state, which is reached by the empty string. It keeps a set of strings S that lead from the initial state to all the states discovered so far. In fact, strings of S represent the states in the Observation Table, which themselves are distinguished according to distinguishing strings from E in the following way: the states represented by two strings s and s' of S are distinguished by a string e of E if e is accepted after s but not after s' or vice versa. S may, and in fact usually will, contain for several states more than one string leading to it. This directly leads to a more abstract representation of states in the Observation Table in terms of a *bit vector* characterizing the acceptance behavior of a state relative to the strings in E.

The L* learning algorithm is characterized by its iterative interplay between membership queries and equivalence queries:

Membership queries are the means to determine the entries of the Observation Table. This is in particular necessary in order to determine the *closedness* and *consistency* of the (hypothesis) automaton currently represented by the Observation Table: after filling the rows for the elements of $S \cdot A$, it can easily be checked whether this automaton is closed under the transition relation, meaning that all successors of all states of this automaton (here represented by strings of S) are themselves represented by elements of S. Technically this means that their rows appear already as a row belonging to some elements of S [cf., Definition 2(1)]. To exploit all the information inherent in the Observation Table, L* additionally checks whether the equivalence relation imposed by the rows onto the elements of S is one-step consistent, meaning that all elements of S with equal rows have equivalent transitions [cf., Definition 2(2)]. If one of these checks fails, L* resolves the discrepancy by extending S or E accordingly.

Closed and consistent (hypothesis) automata are then validated by L* by means of an equivalence query. If the query is not successful, a counterexample is returned in the form of a string that serves to distinguish further states, and a new iteration begins.

Algorithm 4.1 summarizes L* in a high-level notation. The reader may note that L* identifies all states as long as no corresponding distinguishing witness traces have occurred. In that way, L* performs the maximal possible abstraction and leads toward a state-minimal acceptor.

```
Alphabet $A$, Observation Table $\mathcal{OT}$ initialized to
$(S, E) = (\{\lambda\}, \{\lambda\})$

  repeat
    Extend $\mathcal{OT}$ to $(S \cup S \cdot A) \times E$ using membership queries
    while $(\neg isClosed(\mathcal{OT}) \vee \neg isConsistent(\mathcal{OT}))$ do
      if $(\neg isClosed(\mathcal{OT}))$ then
        $\exists s_1 \in S, a \in A.\, \forall s \in S.\, row(s_1 \cdot a) \neq row(s)$
        $S \leftarrow S \cup \{s_1 \cdot a\}$
        Extend $\mathcal{OT}$ to $(S \cup S \cdot A) \times E$ using membership
        queries
      end if
      if $(\neg isConsistent(\mathcal{OT}))$ then
        $\exists s_1, s_2 \in S, a \in A, e \in E.\ row(s_1) = row(s_2) \wedge$
        $\mathcal{OT}(s_1 \cdot a \cdot e) \neq \mathcal{OT}(s_2 \cdot a \cdot e)$
        $E \leftarrow E \cup \{a \cdot e\}$
        Extend $\mathcal{OT}$ to $(S \cup S \cdot A) \times E$ using membership
        queries
      end if
    end while
    $M \leftarrow Conjecture(\mathcal{OT})$
    $\sigma'_c \leftarrow EO(M)$
    if $(\sigma'_c \neq \perp)$ then
      $S \leftarrow S \cup Prefix(\sigma_c)$
    end if
  until $(\sigma'_c = \perp)$
```

Algorithm 4.1: L*

*4.3 Adapting L**

In turns out that it is not too difficult to adapt L* to work with input-deterministic I/O transition systems instead of deterministic finite automata. I/O transition systems differ from ordinary automata in that their edges are labeled with inputs and outputs instead of just one symbol and that there are no accepting or rejecting states. So we do not observe acceptance/rejection, but we observe the output sequences stimulated by inputs.

Dealing with input-deterministic systems, we can reduce access strings to just the sequence of input symbols that lead the system to the given state, omitting the outputs. Also, input determinism permits us to replace the acceptance bit used by L* with the output sequence generated when applying an input to a given state. As a first-hand characterization of a state we take the reactions to single inputs, i.e., the outputs of the transitions originating in the state. Thus we start our adapted Observation Table with one column for each input symbol.

The accordingly modified definition of an Observation Table is given below.

Definition 3 (Observation Table for I/O transition systems). *Given sets A_I and A_O of input and output actions, a finite, prefix-closed set $S \subset A_I^*$ and a finite set $E \subset A_I^+$, an I/O Observation Table $\mathcal{IOOT}$ is a two-dimensional array with rows for each string in $S \cup S \cdot A_I$, columns for strings in E, and entries in A_O^*. An entry $\mathcal{IOOT}(s, e)$ is interpreted as the reaction of the system to be learned to the last element of $s; e$ after applying the sequence to the system in its initial state.*

1. *$\mathcal{IOOT}$ is called **closed** if $\forall t \in S \cdot A_I.\ \exists s \in S.\ row(t) = row(s)$.*
2. *$\mathcal{IOOT}$ is called **consistent** if $\forall s_1, s_2 \in S.\ row(s_1) = row(s_2) \Rightarrow \forall a \in A_I.\ row(s_1; a) = row(s_2; a)$.*

To adapt the learning algorithm accordingly, one has to

– Insert the input actions A_I for A,
– Use $\mathcal{IOOT}$ instead of $\mathcal{OT}$ and
– Initialize the table with $(S, E) = (\{\lambda\}, A_I)$.

These three simple steps transform Algorithm 4.1 into an algorithm that works for I/O transition systems.

4.4 Realizing the membership oracle

Membership queries can be answered by testing the system we want to learn. This is not quite as easy as it sounds, simply 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 (nonpropositional) data. Thus we have to drive the system with real data, which requires reversing the abstraction and produce a concrete stimulation string.

In practice, the reversion of abstraction requires some effort: things abstracted from the observations have to be filled in dynamically, taking the reactions of the system into account. For instance, timestamps have to increase and, instead of symbolic addresses and symbolic tags, their concrete counterparts have to be used consistently. And, finally, all these data have to be transformed into signals and fed to the system.

In the case of our example telephone switch, all this is done by our testing environment, the already mentioned ITE. The ITE performs this task using predefined code blocks for generating stimuli and for capturing responses and glue code, which together solve the problems connected with generating, checking and identifying the nonpropositional protocol elements. Thus much of the work of putting our approach into practice relies on the ITE system and its diverse components.

4.5 Equivalence oracle

Besides the approximations to the equivalence oracle by testing as described in Sect. 4.1, we propose to use formalizations of expert knowledge. An expert – this could be an implementor, a system designer, or an application specialist – provides a specification of an invariant of the system, i.e., a property that all system runs must meet. This specification could be given in some universal temporal logic. We use linear-time temporal logic (LTL, [3]) to formulate such constraints. These constraints can be model checked on a hypothesis automaton.[3]

Preferably, a constraint expresses a safety property, i.e., a property that holds true unless a state is reached that makes it fail. In this case, a failure yields a finite counterexample as displayed at the top of Fig. 3. The last system output in the sequence violates the safety property. Just as a sequence of input stimuli is used to generate a test, the subsequence of input elements of the counterexample can now be fed into the system. There are two possibilities for the observations made during the test:

1. The observed behavior on the system is consistent with the hypothesis automaton. In this case a discrepancy between the specified constraint and the system has been detected. Either the system itself has an error or the specification is wrong. This has to be resolved manually, i.e., by consulting the system or application experts. If the error can be attributed to the constraint, its correction is easy: the constraint is corrected (or dropped), and the learning process can

[3] To be able to do so, either a temporal logic dialect for I/O transition systems has to be used or we have to translate the transition system into a standard finite automaton.

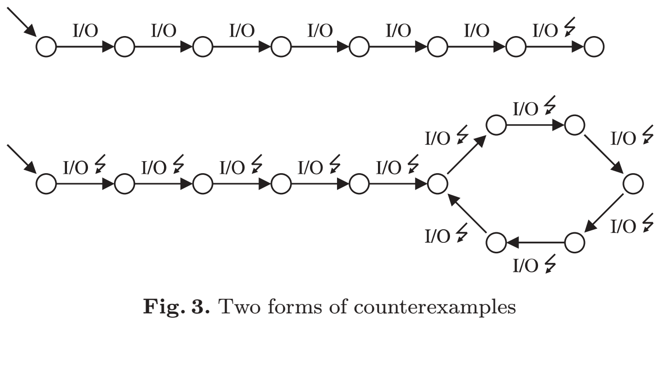

Fig. 3. Two forms of counterexamples

continue. If it is a system error, we found a bug in the system, which is a success in itself. In this case one has to decide manually how to proceed, e.g., whether it makes sense to construct a model of an erroneous system or whether one prefers to correct the error in the model while assuming that the system will be corrected in parallel.
2. The observed behavior deviates from that predicted by the hypothesis automaton. In this case the subsequence is a counterexample as desired from an equivalence oracle. So the learning procedure will take the appropriate actions.

If the constraint is a liveness property, i.e., a property becoming true because some event eventually occurs, counterexamples provided by the model checker will consist of a path leading to a cycle in the hypothesis automaton, so that the infinite sequence resulting from following the path and iterating the cycle contradicts the liveness requirement. The corresponding pattern of such counterexamples is depicted at the bottom of Fig. 3.

It is impossible to test the infinite sequence in the system, but each finite prefix of the sequence yields a test. If some prefix of limited length provides a discrepancy between hypothesis automaton and system, we have a counterexample as in case 2 above, i.e., we were successful in using model checking as a substitute for the equivalence oracle. If not, experts should be consulted to resolve the following questions:

– Was the test horizon large enough for the system to produce some expected reaction which was not observed, or
– Should the length of the tested prefix be increased?

Whereas the appropriate measure is clear in the second case, in the first case, similar to case 1 for safety properties, it has to be manually decided whether the system or the temporal property is wrong, before the appropriate action can be taken.

It should be noted that, although this sounds quite complicated, all these decisions can be taken at the level of the application expert and no knowledge about the implementation of the system or the internals of the model construction machinery is required.

5 Speeding up the learning process

We have adapted the learning procedure to handle I/O transition systems, but there are further structural properties of the systems we want to model that can be exploited for optimization. Some experiences and experimental results with a corresponding implementation are described in [9]. In fact, we observed superlinear efficiency gains when looking at systems with one parametric component (where the number of required membership queries was reduced by a factor of seven) to systems with three parametric components with a factor 100 efficiency gain.

5.1 Pragmatics

It should be noted that L^* as presented in Sect. 4.3 is a theoretical algorithm, which should be tuned for practical use.

For instance, the Observation Table contains redundant information: usually, there will be rows s_1, s_2 and columns e_1, e_2 with $s_1; e_1 = s_2; e_2$. Also, when computing the entries it is best to start with the longest elements from E and fill the entries of prefixes along the way.

Other options for efficiency gains in realizing learning include a lazy approach in filling the entries of the table that only computes entries when they are required. The LearnAutomaton algorithm from [11] can be seen as a lazy variant of L^*. However, this optimization comes at the price of an increased use of the equivalence oracle. Thus its practical impact is difficult to predict.

Another important aspect of practical application is seeding the learning with knowledge about the system behavior. For instance, stimuli sequences initiating known behavior patterns can be entered as elements of E. In this way, sequences that would have to be found by expensive trial and error during the learning phase would be directly available from the start. This is a way to avoid negative effects similar to those of combination locks.

5.2 Symmetries

In our telecommunications example application, the environment may contain several ordinary phones. These phones will be connected to (logical) ports of the switch, and all those ports are supposed to show the same behavior. Therefore, we can abstract from concrete port identifications to abstract names and treat all of them as interchangeable, with various benefits throughout construction and usage of models.

To formally capture the idea of interchangeable components, we use the notion of a *parametric* system $P(n)$, consisting of $n+1$ components $C_0, \ldots, C_n$, where C_0 is meant to be a central component and $C_1, \ldots, C_n$ are peripherals. Let us assume that messages between those components are represented by records that, besides propositional information, reference components by their (symbolic) names $C_1, \ldots, C_n$. An example of such a message is `(hookswitchOnHook,`C_1`)`, where `hookswitchOnHook` indicates the occurred event and the symbolic component name C_1 replaces the concrete address `500` used in the concrete system configuration from Sect. 2.1.

Runs of $P(n)$ consist of sequences of records. We say that the system $P(n)$ is *symmetric* if its set of runs is closed under arbitrary permutations of the $C_1, \ldots, C_n$.

When learning a symmetric system, each observation (membership or equivalence oracle) can be generalized to its symmetric closure, i.e., the set of runs resulting from permutations of $C_1, \ldots, C_n$. If this is done, only symmetrical models are constructed by the learning algorithm and also all hypothesis automata will be symmetrical. Compared to ordinary learning, this results in considerable savings already in standard situations where observations are made on small instantiations like a switch with three phones connected to it.

Further savings result from the related, but different, effects of *independence*: the order in which two records concerning disjoint sets of components without unfinished mutual interactions occur in a run does not matter. This leads to *partial-order* optimizations as discussed below.

5.3 Partial-order methods

This way of capitalizing on specific system properties is inspired from the *partial-order reduction* methods for communicating processes [14, 20]. These methods help to avoid having to examine all possible interleavings among processes when analyzing their behavior. Here, this means that we do not have to learn all interleavings separately.

This works in the following way:

1. An expert specifies explicitly an **independence relation**, e.g., *Two calls with disjoint sets of involved components can be shuffled in any order.*
2. A trace is inspected for independent subparts.
3. Instead of the explicitly observed trace, a whole equivalence class of traces (of which the observed trace is a representative) can be added to the model.

In this way, partial-order methods speed up the model construction and help to keep the model consistent.

Figure 4 shows an example for partial-order equivalence of traces. If there is no connection between C_1 and C_2, `offHook/onHook` events of those phones do not influence each other and can therefore be arranged in any order. The run shown in Fig. 4a contains two such independent `offHook/onHook` sequences: the permutations in Fig. 4b and c are equivalent to those in Fig. 4a and can be added to the model once the first has been observed.

6 Distributed learning

Learning distributed systems like the CTI system above suffers from the well-known state explosion problem: even if the behavior can be modeled by a parallel composition of fairly small automata, the global system model typically is of size exponential in the number of parallel components. The symmetry and partial-order methods mentioned in the previous section are intended to address this concern but fail to provide a general solution.

However, there is a much better alternative if the parallel structure of the system is known and the communications between components are observable: learn the models of the *individual components*. This avoids the state explosion problem and drastically reduces the learning effort from requiring exponentially many membership queries to only a small polynomial number. This is because the complexity of learning by (our variation of) Angluin's algorithm is dominated by the size of the Observation Table, which is in the worst case quadratic in the number of states of the arising hypothesis automata.

Technically, we proceed as follows. We record all internal communications stimulated by external inputs and index each communication by the components involved. From the Observation Table, we keep the row and column labels (i.e., the sets S and E) in their original form and construct smaller tables for all component systems that represent the entries of the original table in more compact form. On this basis, distributed learning mimics Algorithm 4.1 with similar sets of membership and equivalence queries. The smaller tables are built as in the regular process of learning, only we do not strive to fill each entry: permitting a "don't know" value in the small tables, we distinguish states only if there is evidence from incompat-

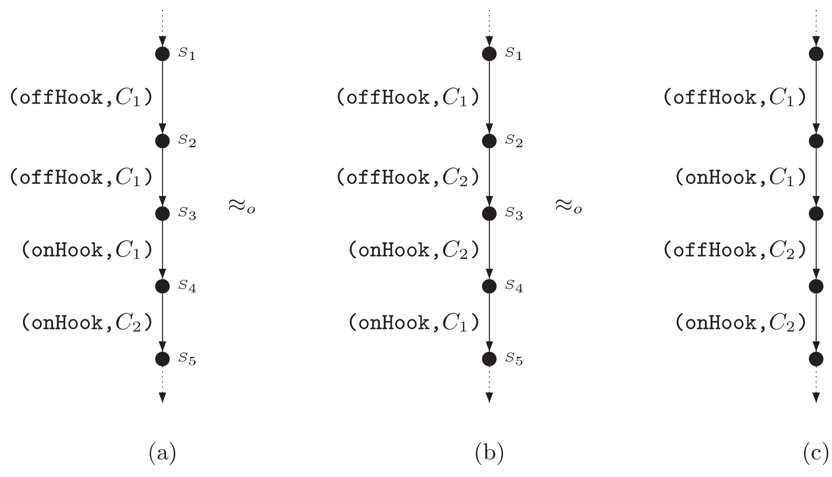

Fig. 4. Examples for reordering

ible observations. This provides us with partial automata tables for the component systems. As these partial tables contain all observations from the global learning process, the result of the learning is not affected by the distributed organization of the Observation Tables.

Stronger assumptions on observability and/or testability may even lead to further performance gains: obviously, if each component can be tested on its own, there is no need for learning the composed system automaton at all. Moreover, it is straightforward to combine distributed learning with other optimization techniques like exploiting properties like prefix closure as well as suppressing redundancy and symmetries.

7 Conclusion and future work

We have discussed behavior-based model construction from a point of view characterized by verification, model checking, and abstraction. It turned out that *abstraction* is the key for scaling known learning techniques for practical applications, that *model checking* may serve as a teaching aid in the learning process underlying the model construction, and that there are also synergies with various validation and *verification* techniques. From the practical perspective, the main issue was to make automata learning, testing, and abstraction compatible. Although these are all very active research areas, they have hardly been considered in combination. At the theoretical level, a notable exception is the work of [5, 17], which proposes a learning-based method for system refinement. Our focus on practicality differs from this work by looking at more rigid abstractions and by exploiting expert knowledge for steering the learning process.

A major problem in behavioral model construction is the growth of the system size and the corresponding complexity of the learning procedures. Thus methods are required to support the construction of aspect-specific models together with the possibility of suppressing redundant information. We are therefore investigating various abstraction techniques and the possibility of constructing concise models by exploiting additional structural information, e.g., concerning the independence of events. First experimental results, which have been reported in [9], demonstrate the impact of our optimizations. In fact, the observed superlinear speedups are already very promising.

Currently we are implementing a distributed learning algorithm along the lines sketched in Sect. 6. This opens up a totally new dimension of optimization with performance gains far beyond the ones reported in [9]. We are therefore convinced that we will be able to scale our techniques to capture essential parts of industrially relevant systems in order to support test-suite design and maintenance, test organization, and test evaluation, as well as the monitoring of running applications.

Acknowledgements. We are very grateful to the ITE team, in particular Oliver Niese, as well as to Tiziana Margaria for discussions and their fruitful comments.

References

1. Angluin D (1987) Learning regular sets from queries and counterexamples. Inf Comput 2(75):87–106
2. Chow TS (1978) Testing software design modeled by finite-state machines. IEEE Trans Softw Eng 4(3):178–187
3. Emerson EA (1990) Temporal and modal logic. In: van Leeuwen J (ed) Handbook of theoretical computer science. Elsevier, Amsterdam
4. Giacobazzi R, Ranzato F, Scozzari F (2000) Making abstract interpretations complete. J ACM 47(2):361–416
5. Groce A, Peled D, Yannakakis M (2002) Adaptive model checking. In: Katoen J-P, Stevens P (eds) Proceedings 8th international conference for tools and algorithms for the construction and analysis of systems, Grenoble, France, April 8–12, 2002. Lecture notes in computer science, vol 2280. Springer, Berlin Heidelberg New York, pp 357–370
6. Hagerer A, Hungar H, Niese O, Steffen B (2002) Model generation by moderated regular extrapolation. In: Kutsche R, Weber H (eds) Proceedings of the 5th international conference on fundamental approaches to software engineering (FASE '02), Grenoble, France, April 8–12, 2002. Lecture notes in computer science, vol 2306. Springer, Berlin Heidelberg New York, pp 80–95
7. Hagerer A, Margaria T, Niese O, Steffen B, Brune G, Ide H (2001) Efficient regression testing of CTI-systems: testing a complex call-center solution. Annual review of communication, vol 55. International Engineering Consortium (IEC), Chicago
8. Hopcroft JE, Ullman JD (1979) Introduction to automata theory, languages, and computation. Addison-Wesley, Reading, MA
9. Hungar H, Niese O, Steffen B (2003) Domain-specific optimization in automata learning. In: Somenzi F, Hunt WA (eds) Proceedings of the 5th international conference on computer aided verification (CAV '03), Boulder, Colorado, USA, July 8–12, 2003. Lecture notes in computer science, vol 2725. Springer, Berlin Heidelberg New York, pp 315–327
10. Hungar H, Steffen B (2003) Behaviour-based model construction. In: Zuck LD, Attie PC, Cortesi A, Mukhopadhyay S (eds) Proceedings of the 4th international conference on verification, model checking and abstract interpretation (VMCAI'03), New York, USA, Jan. 9–11, 2003. Lecture notes in computer science, vol 2575. Springer, Berlin Heidelberg New York, pp 5–19
11. Kearns MJ, Vazirani UV (1994) An introduction to computational learning theory. MIT Press, Cambridge, MA
12. Lang KJ, Pearlmutter BA, Price RA (1998) Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In: Proceedings of the 4th international colloquium on grammatical inference – ICGI '98. Lecture notes in artificial intelligence, vol 1433. Springer, Berlin Heidelberg New York, pp 1–12
13. Lee D, Yannakakis M (1996) Principles and methods of testing finite state machines – a survey. Proc IEEE 84:1090–1123
14. Mazurkiewicz A (1987) Trace theory. In: Brauer W et al (eds) Petri nets, applications and relationship to other models of concurrency. Lecture notes in computer science, vol 255. Springer, Berlin Heidelberg New York, pp 279–324
15. Moore EF (1956) Gedanken-experiments on sequential machines. Ann Math Stud Automata Stud 34:129–153
16. Niese O, Steffen B, Margaria T, Hagerer A, Brune G, Ide H (2001) Library-based design and consistency checks of system-level industrial test cases. In: Hußmann H (ed) Proceedings of the 4th international conference on fundamental approaches to software engineering (FASE '01), Genova, Italy, April 2–6, 2001. Lecture notes in computer science, vol 2029. Springer, Berlin Heidelberg New York, pp 233–248

17. Peled D, Vardi MY, Yannakakis M (1999) Black box checking. In: Wu J, Chanson ST, Gao Q (eds) Proceedings of the joint international conference on formal description techniques for distributed system and communication/protocols and protocol specification, testing and verification (FORTE/PSTV '99), Beijing, China, Oct. 5–8, 1999. Kluwer, Dordrecht, pp 225–240
18. Steffen B, Jay B, Mendler M (1992) Compositional characterization of observable program properties. Int J Theor Comput Sci Appl 26(5):403–424
19. Valiant LG (1984) A theory of the learnable. Commun ACM 27(11):1134–1142
20. Valmari A (1993) On-the-fly verification with stubborn sets. In: Proceedings of the 5th international conference on computer aided verification (CAV '93), Elounda, Greece, June 28–July 1, 1993. Lecture notes in computer science, vol 697. Springer, Berlin Heidelberg New York, pp 397–408
21. Vasilevskii MP (1973) Failure diagnosis of automata. Kibernetika 4:98–108