Int J Softw Tools Technol Transfer (2009) 11:393–407
DOI 10.1007/s10009-009-0111-8

REGULAR PAPER

# LearnLib: a framework for extrapolating behavioral models

**Harald Raffelt · Bernhard Steffen · Therese Berg · Tiziana Margaria**

Published online: 10 April 2009


**Abstract** In this paper, we present the LearnLib, a library of tools for automata learning, which is explicitly designed for the systematic experimental analysis of the profile of available learning algorithms and corresponding optimizations. Its modular structure allows users to configure their own tailored learning scenarios, which exploit specific properties of their envisioned applications. As has been shown earlier, exploiting application-specific structural features enables optimizations that may lead to performance gains of several orders of magnitude, a necessary precondition to make automata learning applicable to realistic scenarios.

**Keywords** Automata learning · Domain-specific optimization · Experimentation · Software library · Grammar inference

## 1 Motivation

Most systems in use today lack adequate specifications or make use of underspecified or even unspecified components. In fact, the much propagated component-based software design style typically leads to under specified 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 telecommunication area: specifications of telecommunication protocols are usually provided as natural language text documents with no formal link to the actual implementation. This hampers the application of any kind of formal validation techniques like model based testing [4] or model checking [30], and it makes it hard to keep them up to date. In fact, in practice very often these specifications are not updated at all, turning the systems all too soon into legacy systems.

Automata learning techniques have been proposed to overcome this situation, by allowing to construct and later update behavioral models automatically [18]. This has been illustrated in the concrete setting of computer telephony integrated (CTI) systems [25,31]. There we observed the system's behavior in its reaction to test sequences. In this setting behavior is understood as the sequences comprising the test stimuli and the corresponding reactions as words of a language over an alphabet consisting of symbols denoting the stimuli and the reactions, respectively. Automata learning aims at constructing finite automata for describing/approximating this language. This technique works well for reactive systems, whenever the chosen interpretation of the stimuli and reactions leads to a deterministic language. For such systems, automata learning can be regarded as *regular extrapolation*, i.e., as a technique to construct the best regular model being consistent with the observations made. This is similar to the well-known polynomial extrapolation, where polynomials are used instead of finite automata, and functions instead of reactive systems. And like there, the quality not the applicability of extrapolation depends on the

H. Raffelt · B. Steffen (✉)
Chair of Programming Systems, TU Dortmund,
Otto-Hahn-Str. 14, 44227 Dortmund, Germany
e-mail: steffen@cs.uni-dortmund.de

H. Raffelt
e-mail: harald.raffelt@uni-dortmund.de

T. Berg
Department of Information Technology, Uppsala University,
751 05 Uppsala, Sweden
e-mail: therese.berg@it.uu.se

T. Margaria
Chair of Services and Software Engineering, Universität Potsdam,
August-Bebel-Str. 89, 14482 Potsdam, Germany
e-mail: margaria@cs.uni-potsdam.de

structure of the considered function. However, due to the enormous degree of freedom inherent in reactive systems, automata learning is computationally much more expensive than polynomial extrapolation. Thus the success of automata learning in practice very much depends on the optimizations employed to exploit the specific profile of the system to be learned, see [27,39] for a successful application scenario.

In this paper, we present the LearnLib, a framework of tools for automata learning, which is explicitly designed for the systematic experimental analysis of the profile of available learning algorithms and corresponding optimizations. Its modular structure allows users to configure their own tailored learning scenarios, which exploit specific properties of their envisioned applications. The ingredients comprise:

- different *model structures*, e.g., deterministic finite automata (DFA) or Mealy machines,
- various *optimization techniques*, in particular a number of structure-driven filters that reduce the structure-dependent redundancy in the learning process,
- two *modes of use*: a *pre-configuration mode* (PCM), which allows the user to pre-configure an optimized learning setting, and a *learning process modelling* mode (LPM), which enable the user to control the entire learning process, comprising the choice of optimizations, strategies of search, as well as the setting of interaction points for a truly interactive learning process, and
- a *simulator* which allows random model generation, in order to support a systematic quantitative analysis of the various constellations.

In particular, the LearnLib is a means to systematically explore the power of optimizations that exploit application-specific structural properties like *the prefix closure* of languages or patterns of *symmetry*. As has been shown earlier [21], exploiting this optimization potential may well be the key to make automata learning applicable to realistic scenarios.

Machine learning techniques have been used in several contexts related to verification and testing [1,7,11,15,16,33] to solve well defined problems. In contrast to these approaches, the LearnLib is not intended to solve a specific problem. It is meant to be an algorithmic core that provides general automata learning methodologies. In principle, each of the approaches mentioned above could have used the LearnLib, and could profit of some of its features. The key idea is that each improvement of the LearnLib algorithms such as domain specific filter techniques (cf. Sect. 5) and strategies (cf. Sect. 5.5) directly leads to improvements of the LearnLib's applications.

## 2 Classical automata learning

Machine learning deals in general with the problem of how to automatically generate system descriptions. Besides the synthesis of static soft- and hardware properties, in particular invariants [5,12,32], the field of *automata learning*, also called regular extrapolation [18] or regular inference [10], is of particular interest for soft- and hardware engineering [8,9,29,33,42].

We have used automata learning techniques in a number of contexts, e.g. to automatically construct models of Web applications as demonstrated in [http://dblp.uni-trier.de/rec/bibtex/conf/issta/RaffeltMSM08] and to enhance incomplete specifications of biological systems [28].

Automata learning tries to construct a deterministic finite automaton that matches the behavior of a given target automaton on the basis of observations of the target automaton and perhaps some further information on its internal structure. The interested reader may refer to [18,38,39] for our view on the use of learning. Here, we only summarize the basic aspects of our realization, which is based on Angluin's learning algorithm $L^*$ from [2].

**Definition 1** A deterministic finite automaton (DFA) is a tuple $\mathsf{M} = (S, s_0, \Sigma, \delta, F)$ where

- $S$ is a finite nonempty set of *states*,
- $s_0 \in S$ is the *initial state*,
- $\Sigma$ is a finite *alphabet*,
- $\delta : S \times \Sigma \to S$ is the *transition function*, and
- $F \subseteq S$ is the set of accepting states.

Intuitively, a DFA evolves through states $s \in S$, and whenever one applies an input symbol (or action) $a \in \Sigma$, the machine moves to a new state according to $\delta(s, a)$. A word $q \in \Sigma^*$ is accepted by the DFA if and only if the DFA reaches an accepting state $s_i \in F$ after processing the word starting from its initial state. We write $s \xrightarrow{a} s'$ to denote that on input symbol $a$ the DFA moves from state $s$ to state $s'$. The transition function $\delta : S \times \Sigma \to S$ can be extended to $\delta' : S \times \Sigma^* \to S$ such that forall states $s, s' \in \Sigma$, letters $a \in \Sigma$, and words $w \in \Sigma^*$ the following holds: $\delta'(s, \varepsilon) = s$, and $\delta'(s, aw) = \delta'(\delta(s, a), w))$.

$L^*$, also referred to as an *active* learning algorithm, learns DFAs by *actively* posing *membership* queries and *equivalence* queries to the target automaton in order to extract behavioral information, and by refining successively an own hypothesis automaton based on the answers. A membership query tests whether a string (a potential run) is contained in the target automaton's language (its set of runs), and an equivalence query compares the hypothesis automaton with the target automaton for language equivalence, in order to determine whether the learning procedure was (already) suc-

cessfully completed. In this case, the experimentation can stop.

In its basic form, $L^*$ starts with the one state hypothesis automaton that treats all words over the considered alphabet (of elementary observations) alike and refines this automaton on the basis of query results iterating two steps. Here, the dual way of how $L^*$ characterizes (and distinguishes) states on its way to construct the minimal deterministic automaton following the pattern of the well-known Nerode congruence is central [20]:

- from *below*, by words reaching them. This characterization is too fine, as different words may well be Nerode congruent, i.e., have the same suffix language. Thus this characterization leads to a relation between states that is contained in the relation corresponding to the Nerode congruence.
- from *above*, by their future behavior wrt. a dynamically increasing finite set of words, which the learning algorithm produces as witnesses of difference wrt. the Nerode congruence: thus future behaviors is essentially characterized by bit vectors, where a '1' means that the corresponding word of the set is guaranteed to lead to an accepting state and a '0' captures the complement. This characterization is typically too coarse, as the considered sets of words are typically rather small, and do not fully capture the Nerode congruence. Thus this characterization leads to a relation between states that contains the relation corresponding to the Nerode congruence.

The second characterization directly defines the hypothesis automata: each occurring bit vector corresponds to one state in the hypothesis automaton, which is successively refined during the learning process.

The initial hypothesis automaton is characterized by the outcome of the membership query for the empty observation. Thus it accepts any word in case the empty word is in the language, and no word otherwise. The learning procedure (1) iteratively establishes local consistency, after which it (2) checks for global consistency.

### 2.1 Local consistency

This first step (also referred to as automatic *model completion*) again iterates two phases: one for checking whether the constructed automaton is *closed* under the one-step transitions, i.e., each transition from each state of the hypothesis automaton ends in a well defined state of this very automaton. And one for checking *consistency* according to the bit vectors characterizing the future behavior as explained above, i.e., whether all reaching words with an identical characterization from above possess the same one-step transitions. If this is not the case, a distinguishing transition is taken as an additional distinguishing future in order to resolve the inconsistency, i.e., the two reaching words with different transition potential are no longer considered to represent the same state.

### 2.2 Global equivalence

After local consistency has been established, an equivalence query checks whether the language of the hypothesis automaton coincides with the language of the target automaton. If this is true, the learning procedure successfully terminates. Otherwise, the equivalence query returns a counterexample, i.e., a word which distinguishes the hypothesis and the target automaton. This counterexample gives rise to a new cycle of modifying the hypothesis automaton and starting the next iteration.

In any practical attempt of learning legacy systems, the equivalence tests can only be approximated, but membership queries can be answered by testing the target systems [18,38]. We investigated several ways for approximating equivalence queries. One such way was via conformance testing [6,13,36,41]. In fact, it turned out that learning and conformance testing have a lot in common [3].

In contrast to DFA's, reactive systems do not distinguish between accepting states and non accepting states, but produce some output in response to the inputs. Mealy machines are well-known models of "deterministic" reactive systems. We therefore adapted Angluin's algorithm to work on Mealy machines [39] in order to better capture the needs of reactive systems.

**Definition 2** A Mealy machine is defined as a tuple $\mathsf{M} = (S, s_0, \Sigma, \Gamma, \delta, \gamma)$ where

- $S$ is a finite nonempty set of *states*,
- $s_0 \in S$ is the *initial state*,
- $\Sigma$ is a finite *input alphabet*,
- $\Gamma$ is a finite *output alphabet*,
- $\delta : S \times \Sigma \to S$ is the *transition function*, and
- $\gamma : S \times \Sigma \to \Gamma$ is the *output function*.

A Mealy machine behaves very similarly to a DFA. It evolves through states $s \in S$, and whenever one applies an input symbol (or action) $a \in \Sigma$, the machine moves to a new state according to $\delta(s, a)$. But in contrast to a DFA it produces an output symbol $x \in \Gamma$ on every move according to $\gamma(s, a)$. We write $s \xrightarrow{a/x} s'$ to denote that on input symbol $a$ the machine moves from state $s$ to state $s'$ producing the output symbol $x$. The transition function $\delta : S \times \Sigma \to S$ can be extended to $\delta' : S \times \Sigma^* \to S$ in the same way as in the DFA case. Also the output function $\gamma : S \times \Sigma \to \Gamma$ can be extended to operate on strings of input symbols by inductively defining $\gamma' : S \times \Sigma^* \to \Gamma^*$ for all states $s, s' \in$

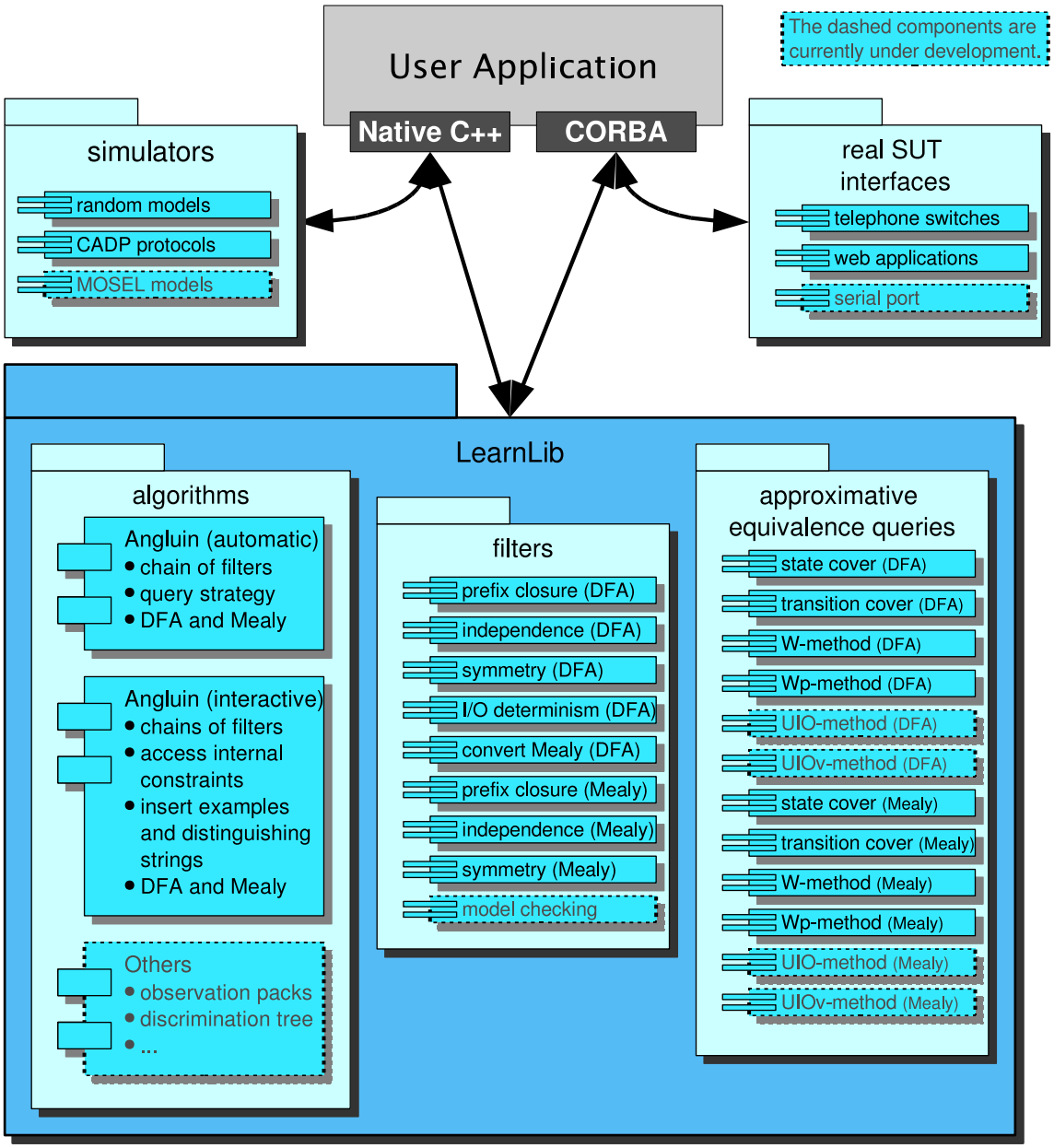


**Fig. 1** LearnLib: components and applications

$S$, letters $a \in \Sigma$ and words $w \in \Sigma^*$ as follows: $\gamma'(s, \varepsilon) = \varepsilon$, and $\gamma'(s, aw) = \gamma(s, a)\gamma'(\delta(s, a), w)$.

## 3 The LearnLib

LearnLib is a framework of tools for automata learning. It is implemented in C++ and tested under Linux and Solaris, and it currently consists of 150 classes and almost 50,000 lines of code. Originally, LearnLib has been designed to systematically build finite state machine models of real world systems. In the meantime, it also became a platform for experimenting with different learning algorithms and to statistically analyze their characteristics in terms of learning effort, run time and memory consumption. As shown in Fig. 1, LearnLib consists of three libraries:

- The *automata learning* framework contains the basic learning algorithms,
- the *filter* library provides several strategies to reduce the number of queries, and
- the *approximative equivalence queries* library is based on the generation of conformance test suites for the conjectures of the learning algorithms.

Applications that use the LearnLib can communicate with the libraries via a CORBA interface or C++ calls.

In the following, we show how to use the LearnLib to generate models of various kinds of legacy systems in Sect. 3.1, and for analysis and profiling in Sect. 3.2.

## 3.1 Model generation for legacy systems

The LearnLib currently provides interfaces to learn models of real-life web applications and of telephony systems. These application domains differ in the art of modeling as well as the interpretation of models.

### *3.1.1 Web applications*

In this scenario, we use the Mealy version of the learning algorithms in the following interpretation:

- The input letters (or actions) of the generated Mealy machine represent HTTP requests (like *open a certain URL*, *submit a form* with some field set to predefined strings, or *follow a specified link*).
- The output symbols in the model represent the HTML pages generated by the web application.

To learn the model of a web application we only need to know the initial URL and the set of strings which should be tried out when submitting forms.

This approach traverses the behavioral structure of the web application by retrieving and analyzing the pages generated by the application. In contrast to web robots, which recursively retrieve all documents that are referenced following only the static link structure of a *website*, in our approach the discovery process is guided by the learning algorithm, which also respects the dynamical behavior of the *web application*. Since all reachable web pages are discovered by and by, and since each reference to a web page is represented by an input letter, we had to extend the learning algorithms to support dynamically growing alphabets. A comprehensive presentation of how our approach is applied to web applications can be found in [35].

In the telephony scenarios [31], which we considered first, we use the DFA version of the learning algorithms and assume a fixed input alphabet.

- The letters of the input alphabet represent simple stimuli like *some phone device lifted the hook or dialed a number*, and checks like *some phone device shows a predefined text in its display*,
- the output alphabet is the set of reactions to those inputs, like *display a certain message*, *ring a certain tone*,
- the membership queries in this scenario are strings of actions of this kind.

Here we define a membership query to be accepting whenever it is possible to stimulate the telephone switch with these inputs and interactions, and none of the checks fails. As in the web application scenario, the abstract membership queries are translated into real test cases which are actually executed on the telephony switch.

Membership queries for such real (legacy) implementations are solved 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) [17,31] 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 non-propositional protocol elements like tags, time stamps and identifiers. In essence, the ITE bridges the gap between the abstract model and the concrete system. The generated test program is then interfaced to the system to be learned with the help of a test harness, comprising both software and hardware.

## 3.2 Analysis and profiling of learning algorithms

Different learning algorithms have different profiles: they differ in the way they proceed to gain structured knowledge about an unknown system. Mostly they differ in the number of membership- and equivalence queries, but also in the size of their queries. In order to analyze these differences and to find out more about how learning algorithms perform in practice we have built a configurator platform and a profiling tool, which allow us to experiment with several learning algorithms and configurations and to collect statistics about their performance under controlled and reproducible experimental conditions. The graphical user interface of the configurator is shown in Fig. 2 in a configuration similar to the one we used in [27] to analyze second order effects among optimizations (described in detail in Sect. 4.1).

A LearnLib configuration is similar to a data flow graph: it specifies how the membership- and equivalence queries generated by one of the LearnLib learning algorithms are passed through other components of the LearnLib, e.g., through optimization filters. Considering Fig. 2, on top of the graph the DFA_Angluin node executes Angluin's algorithm in the DFA version. All the membership queries (MQ) it generates are passed to the (here six) filter configurations, whose (relative) efficiency and performance we want to investigate.

To this aim, the DFA_fork component forwards copies of the queries to six different filter combinations, which eliminate redundancies according to several criteria, as described in detail in Sect. 5. Finally, the queries which could not be filtered away are passed to the system under test for execution by the DFA_SUT_Simulator. This way we analyze the

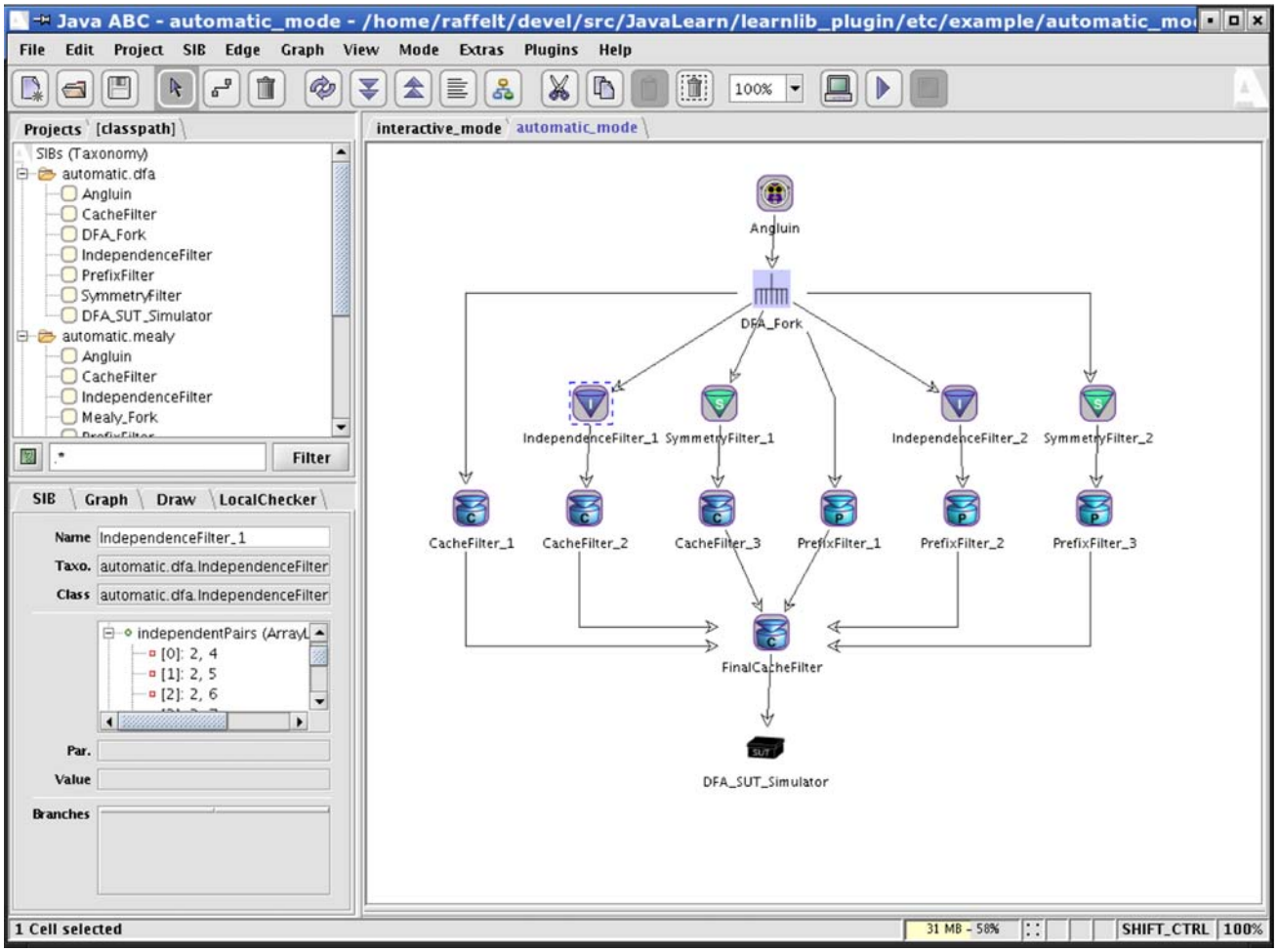


**Fig. 2** Pre-configured learning: designing a filter evaluation setting

**Table 1** Learning CADP-demo protocol specifications

| Protocol | States | Symbols | Indep. actions | Symmetric perm. | Time | Memory | Total MQ | Relevant MQ | Factor % |
|---|---|---|---|---|---|---|---|---|---|
| Petterson | 56 | 18 | 121 | 2 | 1.2 s | 94 k | 43,372 | 1,259 | 2.90 |
| Dekker | 123 | 18 | 115 | 1 | 6.8 s | 214 k | 191,913 | 3,206 | 1.67 |
| POTS | 664 | 32 | 332 | 2 | 394.2 s | 16,564 k | 3,128,115 | 250,721 | 8.02 |

comparative impact of the individual filters while learning a certain model.

For studying the generic behavior of the filters, automatically generated models are most appropriate as they provide us with any required number of example systems. Moreover, the Learnlib allows us to generate models of a particular profile, concerning e.g., the number of accepting or rejecting states, the branching degree, or language features like prefix closure.

In addition to the automatically generated models, we also looked at protocol specifications coming with the CADP toolbox [14]. CADP is a toolbox for the design of communication protocols and distributed systems. We investigated the protocol specification of Dekker's and Petterson's algorithms for mutual exclusion as well as the specification of the Plain Ordinary Telephone System. The results are shown in Table 1.

The table shows our results on three selected instances, which are provided as CADP examples. The specification of Dekker's and Petterson's algorithms have been learned unmodified, and the plain old telephone protocol was adjusted to support exactly two phone devices being able to call each other. The columns states and symbols describe the size of the protocols in the LearnLib's DFA representation.[1] The

[1] Note that the DFA representation requires one more state than CADP's labeled transition system representation in order to realize a sink for all rejected sequences.

inspected models provide a high potential for optimizations, which are described in detail in Sect. 5: the labeled transition systems are by construction prefix closed (cf. Sect. 5.2), the protocols provide a number of independent actions (fourth column and cf. Sect. 5.3), and there are symmetries (fifth column and cf. Sect. 5.4). This leads to significant reductions of the number of membership queries in all three cases.

## 4 Automata learning with the LearnLib

The main library of the LearnLib contains several variants of Angluin's algorithm. Angluin assumes an omniscient oracle (called teacher), which answers the following two kinds of questions (as explained in Sect. 2):

- *Membership queries:* they ask whether a certain word is accepted by the finite state machine. This kind of query can be directly answered for real systems via testing.
- *Equivalence queries:* they ask for checking whether the current conjecture is (already) equivalent to the finite state machine. These queries should be answered either with *Yes* or a *counter example*.

Equivalence queries for 'black box' finite state machines are in general undecidable. Thus one has to live with approximations like e.g., variations of conformance testing, as shown in Fig. 1(right) which lists the conformance testing routines currently available in the LearnLib.

The following two sections describe the two modes offered by the LearnLib to flexibly deal with the wealth of available options: a *pre-configuration mode* (PC-Mode), which allows the user to pre-configure an optimized learning setting (see Sect. 4.1), and a *learning process modeling mode* (LPM-Mode), which enables the user to control the entire learning process, comprising the context-specific choice of optimizations, strategies of search, as well as the setting of interaction points for a truly interactive learning process (see Sect. 4.2).

### 4.1 Pre-configuration mode

In the PC-mode the user graphically specifies a configuration that defines a number of LearnLib experiments carried out in parallel in terms of the chosen learning algorithm, its global selection strategy for membership queries, and the experiment-specific choices of filter chains.

Figure 2 shows the use of the LearnLib as integrated tool to the jABC environment [22]. As we see, a configuration is a data flow graph-like structure which describes how membership- and equivalence queries are passed through components of the LearnLib. This is done by combining the functionalities of the library of components, shown in the upper left frame, to configurations in the canvas on the right. The user of the LearnLib can combine

- a learning algorithm,
- a directed acyclic graph of query filters, to take advantage of structural knowledge about the system,
- a general strategy how to select membership queries, and
- an interface to a system under test.

Configuration design is done, in the usual jABC style, by dragging library components to the canvas, and then connecting them by edges that specify how queries are passed and how the library components should interact. Additionally, each component may have parameters, which are set via the inspector in the lower left panel. In Fig. 2 the parameters of the framed upper left *independence filter* are shown in the lower left panel, which currently indicates that action 2 is independent of actions 4, 5, and 6. The independence filter, described in detail in Sect. 5, is applicable when the system contains pairs of independent actions.

Currently the user can choose between Angluin's method for learning DFAs [2] and our version for learning Mealy machines [39]. To take advantage of knowledge about the analyzed system's structure, the user can specify a chain of filters, discussed in detail in Sect. 5, which are used to reduce the number of membership- and equivalence queries to the oracle (resp. the system under test). The chain of filters must terminate with an oracle or a system under test interface, which answers all unfiltered queries.

Additionally, the user can choose a general strategy of how membership queries are selected, which steers the algorithm either in a more depth or more breadth oriented way. This is particularly interesting in the context of enforcing consistency and closure, where one typically has a variety of options (see also Sect. 5.5). The available strategies so far are

- *random:* choose randomly,
- *fast:* take the first possible alternative,
- *long:* prefer alternatives leading to long membership queries,
- *short:* prefer alternatives leading to short membership queries,
- *cheap:* prefer alternatives producing membership queries that can be answered by the chain of filters, and
- *expensive:* prefer alternatives which produce membership queries that require the membership oracle.

In general the pre-configuration mode of the learn algorithms work as depicted in Fig. 3. An inner loop continues to generate and ask membership queries until the algorithm is able to build a valid hypothesis model. All these membership queries are sent through the chain of filters in order to suppress redundant queries. Queries that remain unfiltered are passed to the membership oracle. This either leads to a direct check (in case of the simulation mode, where the target model is known), or to a test run of the target sys-

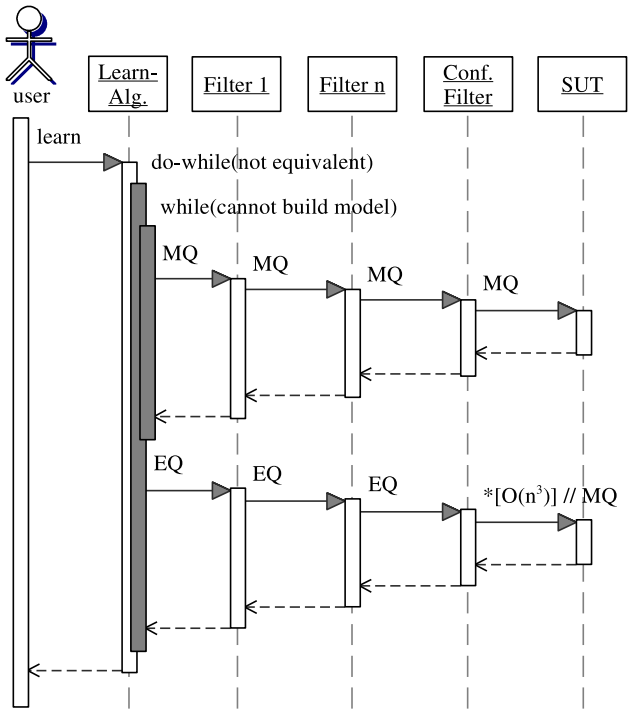


**Fig. 3** Pre-configuration mode

tem (this is the way membership queries are answered in our real life scenarios).

Whenever the learning algorithm has collected enough information to build a valid hypothesis model, this model is subject to an equivalence query, the bottleneck of the learning procedure. Except for the case of simulation, where the target model is known, we have to approximate equivalence queries by means of membership queries of a different kind, that result from a search process for discrepancies. Particularly suitable are here methods adopted from conformance testing (see Sect. 4.3). They help to systematically search for distinguishing execution traces by means of testing, i.e., by posing appropriate membership queries. Thus our filters can also be applied here.

As soon as one of the membership queries detects a discrepancy, the corresponding trace is given to the learning algorithm as a counterexample to improve the hypothesis model, and the next iteration of the learning algorithms begins. This continues until the (approximate) equivalence oracle returns TRUE, signaling that we successfully learned the target system (wrt. this oracle).

### 4.2 Learning process modeling-mode

In the LPM-Mode, graphs which are constructed just as in the PC-Mode, are used to model the entire learning process, which comprises the modeling of conditional or interactive behavior. The nodes may now represent arbitrary statements, in particular including all atomic functionalities of the LearnLib, and the edges specify in which order and under which condition they are processed.

Figure 4 depicts the control flow graph of a simple variation of Angluin's algorithm: here, the execution starts with connecting the graphical user interface to the LearnLib *ConnectToLearnLib*, before an interface to a system under test is created *CreatreSUInterface*. The SUT interface can be linked to a real system, but it can also represent a SUT simulator, which uses known models stored in a database. This means that a SUT interface can represent a number of systems of very different kind. Therefore in the next step it is checked whether there is a next SUT that should be analyzed (*HasNextSubject*).

After this first initialization steps the learning process is started by initializing Angluin's algorithm $L^*$ [2]. The learning algorithm now generates a test suite, which must be executed by the SUT interface in the next step. The *QueryTestCase* component executes the traces contained in the test suite and records the response of the SUT. At this point the SUT interface may discover that the implementation offers more possibilities to be stimulated than currently specified. For example, a new input action may have arisen. This happens for example when learning web applications, since every new discovered web page leads to a new action for directly requesting that page. This special feature is handled by the two components connected to the *sizeChanged* branch. First, the results of querying the SUT are stored in the learning algorithm and then the alphabet is updated. Afterwards the results of querying the SUT are returned as a basis for a user decision *UserInteraction* about the order of the two well-formedness checks *CheckClosure* and *CheckConsistency*. If the observations are both closed and consistent, $L^*$ constructs a conjecture model, which is done in *GetConjecture*, otherwise the learning algorithm provides a new test suite and the main loop continues.

After the main loop, the conjecture can be visualised (*DrawMealyModelGraph*) and stored to a file (*SaveGraph*), before one enters the check for global equivalence. In this example, this is done by generating and then executing a test suite (*CheckTestcases*) according to the Wp-Method [13]. If the conjecture does not conform to the SUT, a counter example is returned, and the learning algorithm continues. Otherwise $L^*$ successfully terminates this learning task and continues with the next SUT (*HasNextSubject*).

The execution of this control flow graph can be interactively steered using the *Tracer*, which is able to execute these control flow graphs. In addition it provides useful debugging functionalities, which allow users to investigate the data exchanged between the nodes resp. atomic functionalities. This way the user can visualize at any time the sets of membership queries and intermediate finite state models gen-

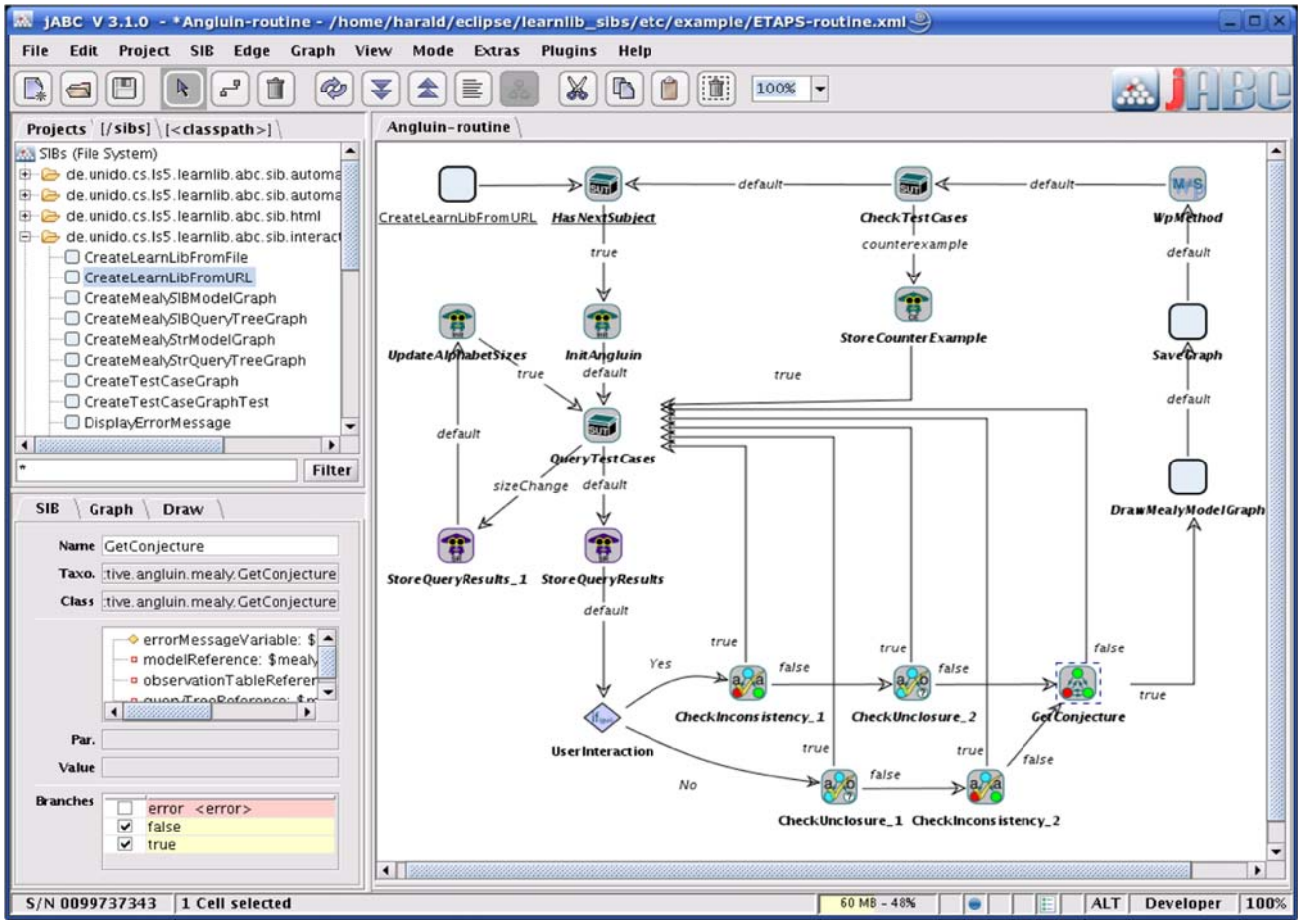


**Fig. 4** LPM Mode: design of an adhoc learning algorithm

erated by the learning process. It is also possible to automatically generate a stand-alone Java program which realizes the specified learning process.

Compared with the PC-mode, the interactive version provides more control on how the learning algorithm proceeds. Like in the PC-mode, learning happens in two alternating phases: constructing hypothesis automata and checking their equivalence with the target automaton. These phases alternate until the equivalence check, which is typically done via some version of conformance testing, is passed. The user however remains in control: at any time he may change the kind of filters used or decide which one of the proposed membership queries should be executed next.

During the first phase this typically happens in five successive steps:

1. Ask the learning algorithm to build a set of membership queries which are required for the learning process. For Angluin's algorithms there are two constraints which must hold before a hypothesis model is built: *closedness* and *consistency* (details can be found [2]). Closedness and consistency are also established via membership queries. In the LPM-Mode, the user may influence the order in which membership queries are posed in order to accelerate the convergence.
2. Decide which filters should be used to filter out irrelevant queries. Chaining of filters is also supported.
3. Send the remaining membership queries to the oracle, which gives the missing answers.
4. Update the filters according to the gained information.
5. Analyze the result of the membership queries given to the learning algorithm, and decide where to continue the iteration.

This loop continues until the learn algorithm is able to construct a hypothesis model for the real system. Now the LearnLib user can use this conjecture for an equivalence query. This is only directly possible in the simulation case, where the target model is known. Otherwise, one typically approximates the equivalence queries by membership queries. Thus we may also profit from the filters here. The LPM-Mode supports this by (Fig. 5)

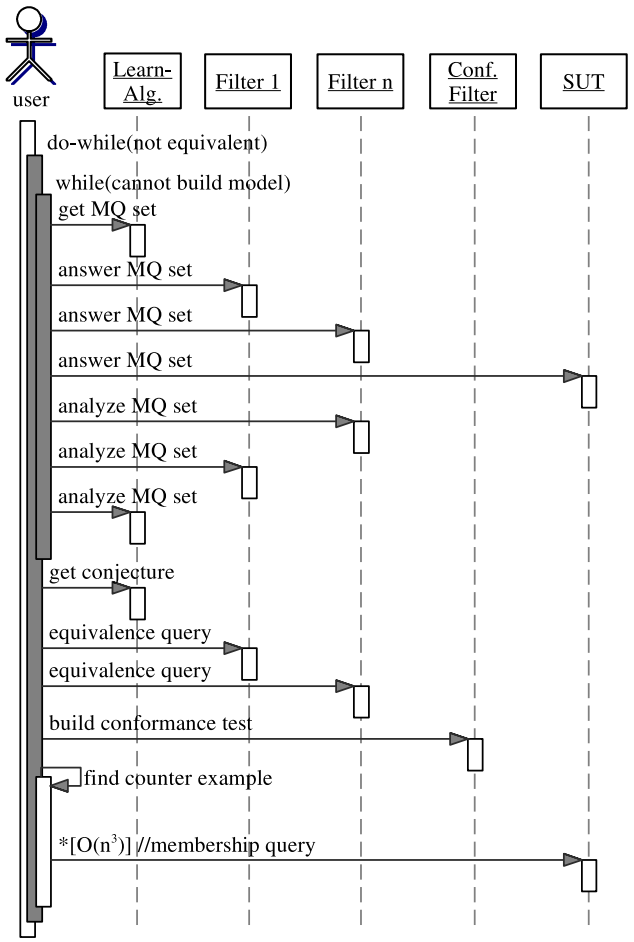


**Fig. 5** Learning process modeling execution model

- allowing to choose specific sets of filters, which may be able to directly produce a counterexample on the basis of the structural assumptions, or
- to translate the conjecture into a (contex-specific) approximating conformance test.

In addition, at any time the user might present the LearnLib with particular execution traces, which he assumes to differentiate the current hypothesis model from the target system. This may drastically reduce the required number of equivalence queries, the true bottleneck of automata learning.

### 4.3 Equivalence queries in the LearnLib

As mentioned before, it is impossible to decide equivalence queries if one is restricted to observe the input/output behavior of an unknown system. Therefore one has to resort to approximations of equivalence queries. As we showed in [28], additional knowledge and counterexamples are necessary here, and it is important to have systematic methods at hand to approximate equivalence queries. It turns out that methods from the field of conformance testing are particularly adequate [3].

The problem of conformance testing can be briefly described as follows [24]. Given

- a finite state machine $M_S$, which acts as known specification, and
- a black-box implementation $M_I$ (typically representing just another finite state machine), providing testing capabilities only,

one wants to determine by testing whether $M_I$ correctly implements or, as we say, *conforms* to $M_S$.

Of course also this problem is in general undecidable, but there are a number of practically relevant approaches, some of which, under certain circumstances, like e.g., restriction of the number of states of the black box implementation, are even complete.

Due to the restricted setting, the proposed alternatives only differ in the set of tests they produce. Thus conformance testing is closely related to test generation. Besides basic test suite generation algorithms like state cover set and transition cover set, the current version of the LearnLib supports also the W-Method [6], the Wp-Method [13], the UIO-method [37], and the UIOv-method [41].

## 5 Optimizations of the LearnLib

Our initial experiments on real systems have shown that test-based learning of systems is impractical: the classical automata learning method automatically generates enormous numbers of MQs which must then be resolved via the execution of test cases. Thus it is important to find ways to drastically reduce the number of these queries. Therefore the LearnLib contains a number of filter techniques which exploit domain and expert knowledge for this purpose. This has led to reductions of several orders of magnitude in realistic scenarios. The interested reader my refer to [27,39] for a detailed exposition.

Figure 6 shows an example of a Mealy machine which will be used as a kind of minimal example to explain the characteristics of the filters. It is not a typical application example for our method, which aims at real life systems like the ones presented in [35,39], but it illustrates in a condensed setting all the essential phenomena that appear there. The machine represents a kind of Turing machine as depicted in Fig. 7, which consists of a write/read head which can move left and right, and a tape of cells. Each cell can hold one value of {*empty*, 0, 1}. In contrast to a Turing machine, the tape has a limited length and its ends are connected like a ring. Initially all cells of the tape are empty. The head can write "0" and "1" but it is impossible to reset a cell to *empty*.

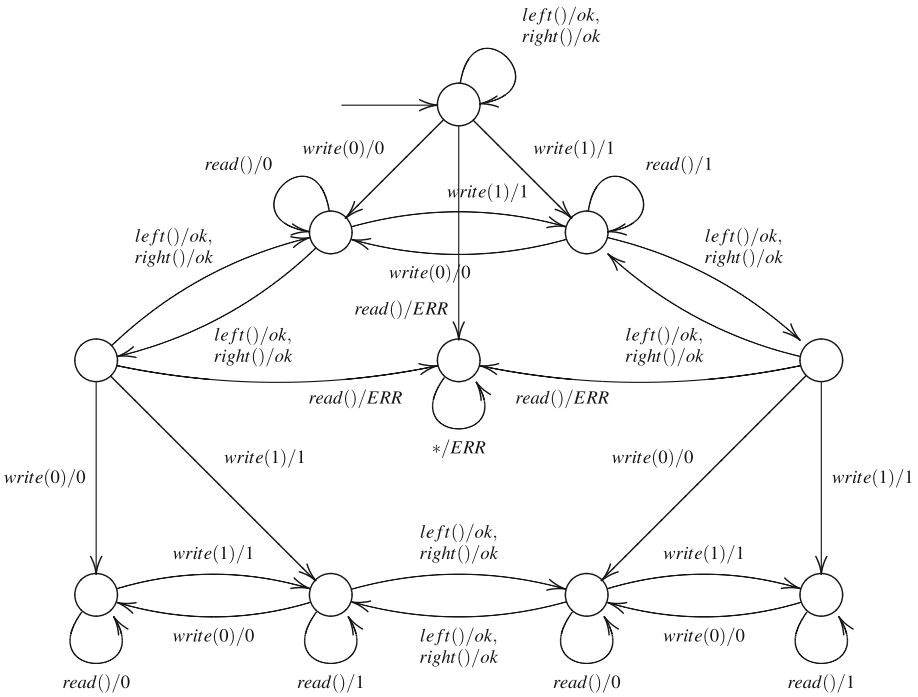


**Fig. 6** Prefix closed Mealy machine with independent actions and symmetry. The input alphabet is {*write*(0), *write*(1), *read*(), *left*(), *right*()} and the output alphabet is {0, 1, *ok*, *ERR*}

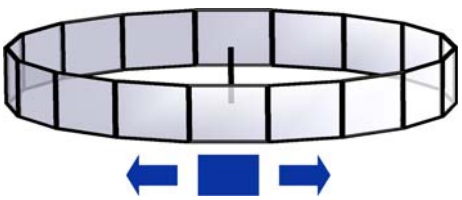

**Fig. 7** A simple Turing-like tape machine

The read operation returns the content of the current cell if it is not *empty*. Otherwise it fails and the machine breaks down, answering any further input with a special error symbol ERR. Figure 6 shows the model of an instance of this machine with tape length 2, which, according to its construction, is prefix closed.

The following sections discuss the main filters. They are ordered according to their degree of specialization.

### 5.1 Redundancy

Angluin's learning algorithm generates redundant membership queries: its central data structure, the observation table [2], which organizes the systematic exploration of the target system by means of membership queries, typically contains numerous equivalent entries, i.e., entries that result from asking the same membership query.

In order to prevent the automated test equipment from executing those test cases repeatedly, a cache is used to detect doubles and filter them out: the cache filter stores every test result in a hash table $T : \Sigma^* \rightarrow \{\text{unknown, true, false}\}$, where three values express the current knowledge about $q$: (1) it has not yet been considered, (2) it is member of the considered language or (3) it is not. Given a specific membership query $\text{MQ}(q)$, the corresponding formal filter rules are straightforward:

- $T(q) = \text{true} \Rightarrow \text{MQ}(q) = \text{true}$
- $T(q) = \text{false} \Rightarrow \text{MQ}(q) = \text{false}$

Only if $T(q) = \text{unknown}$, a test case is required to answer the membership query.

### 5.2 Prefix closure

If the language we want to learn consists of observations of runs of a real system it is obvious that this language is prefix closed, i.e., that given a run, every prefix of this run is also in the language (thus it is itself also a run of the system). This observation leads to a very powerful optimization, as the learning algorithm needs not consider *continuations* of possible runs that have already been excluded from the target language by means of a previous membership query. Also, whenever a long string is known to be a run of the system (this is typically the case when the equivalence query presents a positive counter example, i.e. a run of the system not yet contained in the constructed model), we can add all the prefixes of this string to the model without further testing effort.

Formally, we define:

**Definition 3** A DFA is *prefix closed* if the set of non accepting states $S \backslash F$ is closed under transition.

$$\forall a \in \Sigma.\, \forall s \in (S \backslash F)\,.\, \delta\,(s, a) \in (S \backslash F)$$

In the Mealy case, one can introduce *prefix closure* using a special failure output, which signals that the machine is stuck in a dedicated error state (sink) which cannot be left.

**Definition 4** A Mealy machine is *prefix closed* if there is an output symbol $f \in \Gamma$ such that once the machine produced the failure output $f$, each further response of the machine is $f$ as well.

$$\forall a \in \Sigma.\, \forall s, s' \in S.\, s \xrightarrow{a/f} s' \Longrightarrow \forall b \in \Sigma.\, \gamma(s', b) = f$$

Figure 6 shows the prefix closed Mealy machine to our example system: the failure output symbol is ERR, and once ERR occurred the machine moves to the failure state which is a sink. Note that any minimized prefix closed Mealy machine (resp. DFA) has at most one failure state.

The prefix closure filter is also implemented by means of an optimized cache. The corresponding formal filter rules are straightforward:

- $\exists q' \in \Sigma^*.\, T(q \cdot q') = \text{true} \;\Rightarrow\; \text{MQ}(q) = \text{true}$
- $\exists q' \in \text{prefix}(q).\, T(q') = \text{false} \;\Rightarrow\; \text{MQ}(q) = \text{false}$

### 5.3 Independence of actions

Observable events may be *independent* in the sense that they can be executed in any order leading to the same system state. Thus if we have observed (or queried) one execution order, we can deduce that each reordering of independent events results in the same system state. In particular, if one of these execution orders is a run of the system, then so are all the (equivalent) re-orderings. In the Mealy case it is additionally required that the output sequence of an execution is permuted according to the reordering of the input sequence. The independence filter exploits this observation by only querying the system for *one member* of each such equivalence class.

In contrast to the prefix closure filter, the independence filter requires the input of an application domain expert in form of an independence relation that specifies which events can be *shuffled* in any order.

As an example, the Mealy machine in Fig. 6 contains one pair of independent actions: (*left*(), *right*()). Note that this is the only pair of independent actions, which means that the machine only reaches the same system state, producing an equivalent output sequence, when the *left*() action is directly followed by the *right*() action. A *write* action and even a *read* action in between may result in a different system state or an inequivalent output sequence.

Formally, independence is an irreflexive and symmetric relation on pairs of actions.

**Definition 5** Two actions $a, b \in \Sigma$ of a minimal DFA are independent, if and only if in every state of the system the input sequences $a; b$ and $b; a$ lead to the same successor state.

$$\forall a, b \in \Sigma.\, \forall s \in S.$$
$$\delta(\delta(s, a), b) = \delta(\delta(s, b), a)$$

**Definition 6** Two actions $a, b \in \Sigma$ of a minimal Mealy machine are independent, if and only if they are independent in the sense of Definition 5 and the output is permuted accordingly.

$$\forall a, b \in \Sigma.\, \forall s \in S.$$
$$\delta(\delta(s, a), b) = \delta(\delta(s, b), a) \wedge$$
$$\gamma(s, a) = \gamma(\delta(s, b), a) \wedge \gamma(s, b) = \gamma(\delta(s, a), b)$$

The independence relation induces an equivalence relation $\equiv_I \subseteq \Sigma^* \times \Sigma^*$ on the queries, whereby two queries $q$ and $q'$ are equivalent if and only if there exists a reordering of the events conform to the independence relation that transforms the query $q$ into $q'$.

Our independence filter *normalizes* queries according to the independence relation: it calculates the lexicographical smallest equivalent query based an a given ordering on the actions.

### 5.4 Symmetry

Hardware and also telecommunication systems often contain large numbers of components which are instances of a same kind. They cannot be distinguished from each other without explicitly looking at their identification number.

From an observational point of view, it often does not matter *which* device of a certain kind is performing a certain action (e.g., which memory bank is addressed, or which phone calls a certain number), and also the precise identification of the counterpart (the requesting processor or the receiver of the call) is in principle unimportant, as long as we assume a unique and consistent identification, e.g., that the called number and the number of the receiver match.

This observation provides an enormous optimization potential which drastically grows with the number of identical components in a system. We implemented a corresponding filter, which in its essence leads to a symbolic treatment of the devices: we number the *actors* (processors, memories, phones) according to their appearance in a particular run, and we match runs according to this numbering. Moreover, the symbolic numbers are 'freed' whenever the corresponding actor reaches its initial state again. The resulting model is essentially as complicated as the real world scenario with $n$ actors (of a kind), where $n$ is the maximal number of actors

**Table 2** Valid permutations of the Mealy machine in Fig. 6

| Input permutation | Output permutation |
|---|---|
| *id* | *id* |
| (*write*(0), *write*(1)) | (0, 1) |
| (*left*(), *right*()) | *id* |
| (*write*(0), *write*(1)) (*left*(), *right*()) | (0, 1) |

being active (not idle) in the model at the same time. Like for independence of actions, it is the expert who determines which devices are considered equivalent in the sense above.

Formally this characteristic can be described as a permutation group over the alphabet. Each permutation describes a valid interchange of actions under which the system behaves symmetrically. In the DFA case it is sufficient to describe how the (input) actions can be permuted. In the Mealy case a permutation of input actions usually requires the output actions to be permuted as well. For example, when exchanging the action *write*(0) by *write*(1) (and vice versa) in the Mealy machine of Fig. 6, one also has to exchange the output 0 by 1 (and vice versa) to gain an equivalent model. Table 2 contains the list of valid permutations for this example.

**Definition 7** A permutation $\pi$ over the alphabet $\Sigma$ is valid for a deterministic finite state machine $\mathsf{M} = (S, s_0, \Sigma, \delta, F)$, if it maps the alphabet so that each word of mapped symbols is accepted by the DFA if and only if the original word is also accepted.

$$\forall w \in \Sigma^*.\ \delta(s_0, w) \in F \Leftrightarrow \delta(s_0, \pi(w)) \in F$$

**Definition 8** A pair of permutations $(\pi_\Sigma, \pi_\Gamma)$ over the alphabets $\Sigma$ and $\Gamma$ is admissible for a Mealy machine $\mathsf{M} = (S, s_0, \Sigma, \Gamma, \delta, \gamma)$, if the following holds.

$$\forall w \in \Sigma^*.\ \gamma(s_0, w) == \pi_\Gamma(\gamma(s_0, \pi_\Sigma(w)))$$

In other words, the input permutation $\pi_\Sigma$, the output permutation $\pi_\Gamma$, and the output function $\gamma$ have to commute according to the diagram depicted in Fig. 8.

The implementation of the symmetry filter normalizes the queries as well. This is done by choosing a permutation which maps a given query to the lexicographically smallest equivalent query. In contrast to the independence filter, which is local in the sense that it shuffles single actions on a query, the scope of the symmetry filter is global: it acts on the whole context of a query.

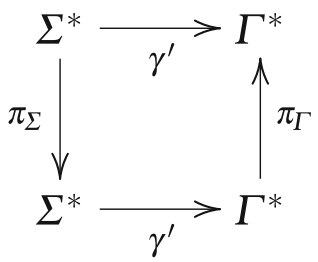


**Fig. 8** Commutative diagram of Mealy symmetry

### 5.5 Choice of strategies

Typically there are numerous ways of enforcing consistency and closure. We therefore investigated the impact of various strategies for choosing appropriate membership queries to this aim. The diagram in Fig. 9 shows the influence of this choice upon learning randomly generated DFAs. Each bar in the figure represents the average number of membership queries resulting from 1,000 experiments.

The DFAs used in this simulation have 100 states and 64 actions each, but the number of accepting states varies between 1 and 50. The entries in the legend represent which combination of strategies are used to enforce *consistencies* and *closedness*. The considered strategies analyze the required membership queries for each alternative and choose one accordingly. For example, the strategies with predicate *short* take inconsistency first, which requires a short membership query, while other prefer longer membership queries (long), membership queries that can be answered without additional test (fast), or they just choose randomly (random). A more detailed description of this particular experiment goes beyond the scope of this paper, which only intends to provide an impression of the features the LearnLib provides to its users.

Note that it is not necessary to experiment with 50,…,99 accepting states when using completely random generated models, since accepting and rejecting states are handled symmetrically. The results show that at least the strategies for choosing inconsistencies have a large impact on the number of necessary membership queries. But they also show that there is no optimal strategy covering all situations: the choice depends on the structure of the model. For example, preferring short membership queries on DFAs with a balanced number of accepting and rejecting states leads to quite good results, but in the unbalanced case one should prioritize longer membership queries.

## 6 Conclusion

In this paper, we have presented the LearnLib, a modular framework for automata learning, which is explicitly designed for experimentation with a variety of learning technologies and tools, in the context of different application scenarios. Its modular structure allows users to systematically analyze and then construct learning algorithms tailored for their specific application scenario. The power of such a

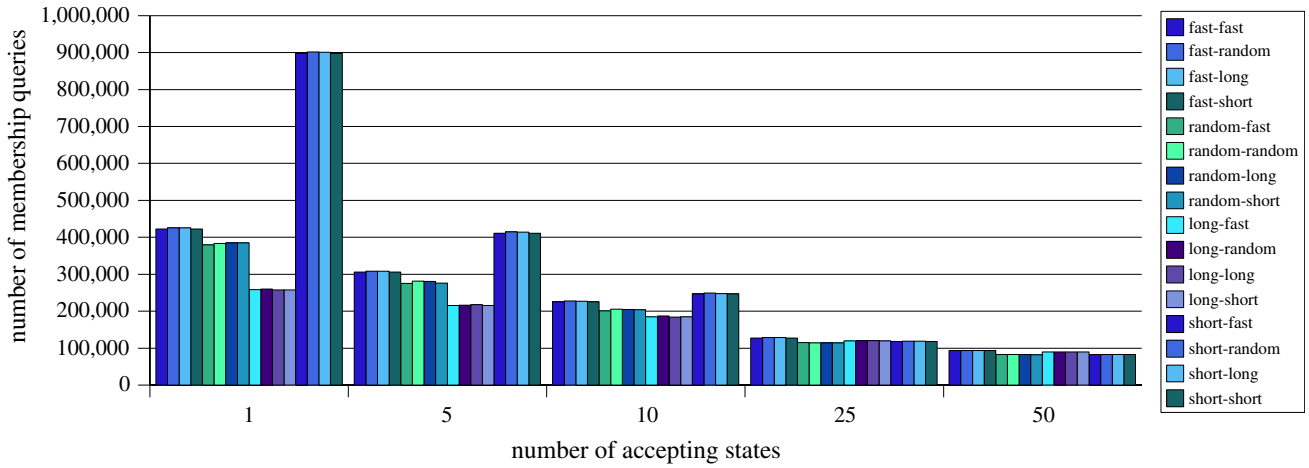


**Fig. 9** Influence of strategy on random generated DFAs (100 states and 64 actions)

tailoring has been illustrated earlier in the concrete setting of computer telephony integrated (CTI) systems, with a performance gain of several orders of magnitude.

It turned out that the variety of options for optimizing and steering learning algorithms and their impact is vast, and that each new experimental analysis revealed new and unforseen insights. We therefore envisage the extension of our framework by an increasing number of functionalities, exploiting other specific structural properties, like e.g., system architecture, other phenomena, like real time, and increasingly complex application scenarios. The extension of the LearnLib as well as the experimental work are very labour intensive. Meanwhile we have made it available over the Internet by means of the jETI technology [New1] [26,40], and it has been already used by other groups in order to create their own learning scenario settings [New2].

As the extension of the LearnLib as well as the experimental work are very labor intensive, we are planning to make the LearnLib available over the Internet by means of the jETI technology [26,40].

## References


1. Alur, R., Cerny, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. In: POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 98–109. ACM Press, New York, NY, USA (2005)
2. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **2**(75), 87–106 (1987)
3. Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., Steffen, B.: On the correspondence between conformance testing and regular inference. In: Cerioli, M. (ed.) Proceedings of 8th International Conference on Fundamental Approaches to Software Engineering (FASE'05), LNCS, vol. 3442, pp. 175–189. Springer, New York (2005)
4. Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A.: Model-based Testing of Reactive Systems, LNCS, vol. 3472. Springer, New York (2005)
5. Brun, Y., Ernst, M.D.: Finding latent code errors via machine learning over program executions. In: Proceedings of the 26th International Conference on Software Engineering (ICSE'04), pp. 480–490. Edinburgh, Scotland (2004)
6. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. **4**(3), 178–187 (1978)
7. Cobleigh, J.M., Giannakopoulou, D., Pasareanu, C.S.: Learning assumptions for compositional verification. In: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003), LNCS, vol. 2619, pp. 331–346. Springer, Berlin/Heidelberg (2003)
8. Cook, J.E., Wolf, A.L.: Discovering models of software processes from event-based data. (TOSEM) ACM Trans. Softw. Eng. Methodol. **7**(3), 215–249 (1998)
9. Cook, J.E., Du, Z., Liu, C., Wolf, A.L.: Discovering models of behavior for concurrent systems. Technical Report, New Mexico State University, Deppartment of Computer Science. NMSU-CS-2002-010 (2002)
10. de la Higuera, C.: A bibliographical study of grammatical inference. Pattern Recognit. **38**, 1332–1348 (2005)
11. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering **27**(2), 1–25 (2001). A previous version appeared in ICSE '99, Proceedings of the 21st International Conference on Software Engineering, pp. 213–224. Los Angeles, CA, USA, May 19–21 (1999)
12. Ernst, M.D., Czeisler, A., Griswold, W.G., Notkin, D.: Quickly detecting relevant program invariants. In: Proceedings of 22nd International Conference on Software Engineering (ICSE'00), pp. 449–458 (2000)
13. Fujiwara, S., von Bochmann, G., Khendek, F., Amalou, M., Ghedamsi, A.: Test selection based on finite state models. IEEE Trans. Softw. Eng. **17**(6), 591–603 (1991)
14. Garavel, H.: Open/caesar: an open software architecture for verification, simulation, and testing. In: Steffen, B. (ed.) Proceedings of the 1st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), LNCS, vol. 1384, pp. 68–84. Springer, New York (1998)
15. Groce, A., Peled, D., Yannakakis, M.: Adaptive model checking. In: Katoen, J.P., Stevens, P. (eds.) Proceedings of the 8th Internation Conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 2280, pp. 357–370. Springer, New York (2002)
16. Habermehl, P., Vojnar, T.: Regular model checking using inference of regular languages. In: Proceedings of 6th International

Workshop on Verification of Infinite State Systems (INFINITY 2004), Electronic Notes in Theoretical Computer Science, vol. 138, pp. 21–36. Elsevier Science (2005)
17. Hagerer, A., Margaria, T., Niese, O., Steffen, B., Brune, G., Ide, H.D.: Efficient regression testing of cti-systems: Testing a complex call-center solution. Annu. Rev. Commun. Int. Eng. Consort. (IEC), Chicago (USA) **55**, 1033–1040 (2001)
18. Hagerer, A., Hungar, H., Niese, O., Steffen, B.: Model generation by moderated regular extrapolation. In: Kutsche, H.W.R. (ed.) Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE'02), LNCS, vol. 2306, pp. 80–95. Springer, Heidelberg, Germany (2002)
19. http://dblp.uni-trier.de/rec/bibtex/conf/iceccs/MargariaRSL07
20. Hungar, H., Steffen, B.: Behavior-based model construction. Int. J. Softw. Tools Technol. Transf. (STTT) **6**(1), 4–14 (2004)
21. Hungar, H., Margaria, T., Steffen, B.: Test-based model generation for legacy systems. In: Proceedings of 2003 International Test Conference (ITC 2003), pp. 971–980. IEEE Computer Society, Charlotte, NC (2003)
22. Jörges, S., Kubczak, C., Nagel, R., Margaria, T., Steffen, B.: Model-driven development with the jabc. In: Proceedings of Haifa verification conference 2006 (HVC 2006), LNCS, vol. 4383, pp. 92–108. Springer, Berlin/Heidelberg (2007)
23. Kubczak, C., Margaria, T., Nagel, R., Steffen, B.: Plug and play with FMICS-jETI: beyond scripting and coding. ERCIM News N. 73, April 2008, pp. 41–42. http://ercim-news.ercim.org/content/view/346/539/
24. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines—a survey. Proc. IEEE **84**(8), 1090–1126 (1996)
25. Margaria, T., Niese, O., Steffen, B., Erochok, A.: System level testing of virtual switch (re-)configuration over ip. In: Proceedings of the IEEE European Test Workshop (ETW'02), pp. 67–74. IEEE Computer Society Press (2002). ETW2002
26. Margaria, T., Nagel, R., Steffen, B.: Remote integration and coordination of verification tools in JETI. In: Proceedings of the 12th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2005), pp. 431–436. IEEE Computer Society (2005)
27. Margaria, T., Raffelt, H., Steffen, B.: Analyzing second-order effects between optimizations for system-level test-based model generation. In: Proceedings of IEEE International Test Conference (ITC'05), pp. 7, 467. IEEE Computer Society (2005)
28. Margaria, T., Hinchey, M.G., Raffelt, H., Rash, J., Rouff, C.A., Steffen, B.: Completing and adapting models of biological processes. In: Proceedings of IFIP Conference on Biologically Inspired Cooperative Computing (BiCC 2006), Santiago (Chile), pp. 43–54. Springer (2006)
29. Mariani, L., Pezzè, M.: A technique for verifying component-based software. In: Proceedings of Interantional Workshop on Test and Analysis of Component Based Systems (TACoS'04), pp. 17–30 (2004)
30. Müller-Olm, M., Schmidt, D., Steffen, B.: Model-checking: a tutorial introduction. In: Cortesi, G.F.A. (ed.) Proceedings of Static Analysis Symposium (SAS'99), Venice, Italy, LNCS, vol. 1694, pp. 330–354. Springer, Heidelberg, Germany (1999)
31. Niese, O., Steffen, B., Margaria, T., Hagerer, A., Brune, G., Ide, H.D.: Library-based design and consistency checking of system-level industrial test cases. In: Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering (FASE '01), LNCS, vol. 2029, pp. 233–248. Springer, London, UK (2001)
32. Nimmer, J.W., Ernst, M.D.: Automatic generation of program specifications. In: Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA'02), pp. 229–239. Rome, Italy (2002)
33. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: Wu, J., Chanson, S.T., 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: pp. 225–240. Kluwer Academic Publishers (1999)
34. Raffelt, H., Steffen, B.: Learnlib: A library for automata learning and experimentation. In: Baresi, L., Heckel, R. (eds.) Proceedings of 9th International Conference on Fundamental Approaches to Software Engineering (FASE 2006), LNCS, vol. 3922, pp. 377–380. Springer (2006)
35. Raffelt, H., Steffen, B., Margaria, T.: Dynamic testing via automata learning. In: Proceedings of the Haifa Verification Conference 2007 (HVC '07), LNCS, vol. 4899, pp. 136–152. Springer, Berlin, Heidelberg (2008)
36. Sabnani, K., Dahbura, A.: A protocol test generation procedure. Comput. Netw. ISDN Syst. **15**(4), 285–297 (1988)
37. Shen, Y.N., Lombardi, F., Dahbura, A.T.: Protocol conformance testing using multiple uio sequences. In: Proceedings of the 9th International Symposium on Protocol Specification, Testing and Verification, pp. 131–143. North-Holland (1990)
38. Steffen, B., Hungar, H.: Behavior-based model construction. In: Mukhopadhyay, S., Zuck, L. (eds.) Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'03), LNCS, vol. 2575, pp. 5–19. Springer (2003)
39. Steffen, B., Margaria, T., Raffelt, H., Niese, O.: Efficient test-based model generation of legacy systems. In: Proceedings of the 9th IEEE International Workshop on High Level Design Validation and Test (HLDVT'04), pp. 95–100. IEEE Computer Society Press, Sonoma, CA, USA (2004)
40. Steffen, B., Margaria, T., Nagel, R.: jETI: A tool for remote tool integration. In: Halbwachs, N., Zuck, L.D. (eds.) Proceedings of 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05):, LNCS, vol. 3440. Springer, Edinburgh, UK (2005)
41. Vuong, S., Chan, W., Ito, M.: The UIOv-method for protocol test sequence generation. In: de Meer, J., Machert, L., Effelsberg, W. (eds.) Proceedings of 2nd International Workshop on Protocol Testing Systems (IWPTS'89), pp. 161–175. North-Holland (1990)
42. Xie, T., Notkin, D.: Mutually enhancing test generation and specification inference. In: Petrenko, A., Ulrich, A. (eds.) Proceedings of 3rd International Workshop on Formal Approaches to Testing of Software (FATES'03), LNCS, vol. 2931, pp. 60–69. Springer (2004)