Innovations Syst Softw Eng (2005) 1: 147–156
DOI 10.1007/s11334-005-0016-y

ORIGINAL PAPER

Tiziana Margaria · Harald Raffelt · Bernhard Steffen

Knowledge-based relevance filtering for efficient system-level test-based model generation

Received: 4 April 2005 / Accepted: 12 June 2005 / Published online: 29 July 2005

Abstract Test-based model generation by classical automata learning is very expensive. It requires an impractically large number of queries to the system, each of which must be implemented as a system-level test case. Key in the tractability of observation-based model generation are powerful optimizations exploiting different kinds of expert knowledge in order to drastically reduce the number of required queries, and thus the testing effort. In this paper, we present a thorough experimental analysis of the second-order effects between such optimizations in order to maximize their combined impact.

1 Motivation

Validating complex heterogeneous systems escapes established formal approaches, both for system testing and for design verification. Characteristic here in fact is the lack of (formal or semiformal) *operational models* for many of the hardware and software systems constitute such a scenario. Typical application domains are telecommunication systems and systems on a chip, but also EAI (enterprise application integration) scenarios, which are large software integration projects suffering from an almost complete lack of usable models for the components' behaviors (think, e.g., of an SAP installation with all the customization modules!). This situation is not only due to a lack of care in the production and maintenance of up-to-date models of all the system components. Rather, there is a policy of *hiding* intellectual property, since the operational models of the commercial products would reveal too much about protected techniques and designs. Thus the lack of models and the need to "discover" them postproduction will accompany system-level design and testing for a long time to come.

Particularly typical in practice are systems that, additionally, include different commercial components that come from various producers and are made available as products, without any model. Our direct industrial experience stems from the area of Computer Telephony Integrated (CTI) systems. In the past we developed a piece of automated testing equipment (ITE) [14, 18] that has been used for industrial system-level testing of over 200 COTS applications that interoperate with a family of midrange telecommunication switches. Characteristic here was the absence of any form of (formal or semiformal) *operational model* for the hardware and software systems constitute a CTI scenario, which are therefore seen and treated as black boxes. In particular, there is no basis for test coverage considerations, focused test suite enhancement, or systematic maintenance support.

Fortunately one can observe that the models one needs to "discover" in order to profitably *use* the components are far smaller than the full model of the implementation: they are much more abstract, since they provide information about only the interface behavior exposed to the environment. In this respect, complexity experts often talk of *cognitive* complexity (i.e., what a user needs to know in order to use a product), which is much smaller than the *technical* or *intensional* complexity, which is the complexity of the design. Adequate cognitive models are widely considered to be sufficient for interface systems (Web Services technology and model-driven design development are based entirely on this assumption). A key enabling factor for such approaches is an efficient technique for producing cognitive models of systems that relies on system-level testing.

A pragmatic yet theoretically well-founded and systematic solution was proposed in [10, 15], where we presented a method for synthesizing *expressive hypothesis models* from observations of a system. The central idea was to exploit our preexisting automated testing machinery [13, 18] as a system

T. Margaria (✉)
Chair of Service Engineering for Distributed Systems,
Universität Göttingen, Germany,
E-mail: margaria@cs.uni-goettingen.de

H. Raffelt · B. Steffen
Lehrstuhl für Programmiersysteme,
Universität Dortmund, Germany ,
E-mail: (steffen,raffelt)@ls5.cs.uni-dortmund.de,

T. Margaria · B. Steffen
METAFrame Technologies GmbH, Dortmund (Germany),
E-mail: info@metaframe.de

observation tool for constructing models via (adapted) classical automata learning algorithms [24].

Classical automata learning algorithms, like L^* [1], experiment blindly with a system: they generate a huge number of test cases that must be executed in order to build structured models of the system. This is inefficient and impracticable for real industrial systems as it requires executing a huge number of test cases, only some of which bring more knowledge about the system's behavior. Optimizations that use expert knowledge [16,27] are very useful in filtering out irrelevant test cases [15]: these studies have shown that a significant efficiency gain is possible (measured factors of 400 for some examples), but the question is how to optimally exploit that knowledge in a systematic way. Key in the tractability of observation-based model generation are in fact powerful optimizations exploiting different kinds of *expert knowledge* to drastically reduce the number of required experiments and, thus, the testing effort connected with model learning. In this paper, we present a thorough analysis of the efficiency as well as the dependencies and mutual influences (called *second-order effects* in the program optimization community) between such optimizations in order to maximize their combined impact.

The paper is organized as follows. We recall classical automata learning in Sect. 2, describe our use of expert knowledge to filter out irrelevant experiments in Sect. 3, and present their impact on a practical application in Sect. 4. Then, Sect. 5 sets the scene for our systematic statistical analysis before we analyze the impact of the relevant filter compositions in Sect. 6 and their mutual second-order effects in Sect. 7. Finally, Sect. 8 presents our discussion of related work before we conclude in Sect. 9.

2 Background: classical automata learning

Machine learning deals in general with the problem of how to automatically generate a system's description. Besides the synthesis of static soft- and hardware properties, in particular invariants [3,7,20], the field of *automata learning* is of particular interest for soft- and hardware engineering [5,6,19,21,26].

Automata learning tries to construct a deterministic finite automaton (see below) 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 [10,24,27] 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* [1].

Definition 1 *A deterministic finite automata (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,*
- Σ *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 $s \in S$*, but 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.*

L* learns a finite automaton by posing *membership* and *equivalence* queries to that automaton in order to extract behavioral information and refining successively its own hypothesis automaton based on the answers. A membership query tests whether a string (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 to determine whether the learning procedure was (already) successfully completed and the experimentation can be terminated.

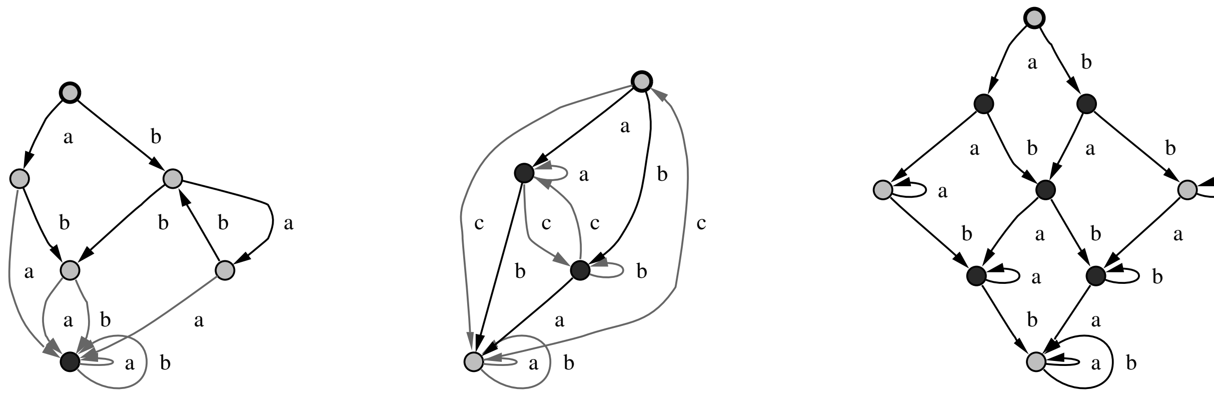
In its basic form, L* starts with the one-state hypothesis automaton that treats all words over the considered alphabet (of elementary observations) alike and refines this automaton on the basis of query results iterating two steps. Here, L*'s dual way of characterizing (and distinguishing) states is central:

- From *below*, by words reaching the states. This characterization is too fine, as different words may well lead to the same state.
- From *above*, by the states' future behavior w.r.t. a dynamically increasing set of words. These future behaviors are essentially 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.

The second characterization directly defines the hypothesis automaton: each occurring bit vector corresponds to one state in the hypothesis automaton.

The initial hypothesis automation 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 state otherwise. Now the learning procedure iteratively establishes local consistency after which it checks for global consistency.

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



(a) Prefix Closed DFA (b) Independent Actions a, b (c) Symmetric Actions a, b

Fig. 1 Deterministic finite state machines with different characteristics

different transition potentials are no longer considered to represent the same state.

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 it does, the learning procedure successfully terminates. Otherwise, the equivalence query returns a counterexample, i.e., a word that 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 at learning legacy systems, equivalence tests can only be approximated (which we will not explain here), but membership queries can often be answered by testing [10,24].

3 Exploiting knowledge to filter irrelevant queries

The learning algorithm uses membership queries extensively to systematically explore a system's behavior, until no more "direct evidence" of inconsistencies between an updated model and the system behavior is detected. It is the goal of our work to drastically reduce the number of these queries in order to open this method to realistic systems. We attack this problem by exploiting domain and expert knowledge. In fact, already four observations on the structure of a system's behavior, which hold for most practical reactive systems, lead to the definition of powerful filters on the queries generated by L*.

In the following subsection, we briefly discuss these filters in order of increasing specialization.

3.1 Redundancy (C-filter)

In the classical implementation, which systematically explores a system's behavioral capabilities, Angluin's learning algorithm may generate redundant membership queries, i.e., produce different derivations for the same test case. In order to prevent the automated test equipment from executing those test cases twice, a cache is used to detect doubles and filter them out: the C-filter stores every test result in a hash table $T : \Sigma^* \rightarrow \{\text{true, false, unknown}\}$, where the three-valued truth value $T(q)$ expresses the current knowledge about whether q is a member of the considered language or not. The corresponding formal filter rules applied to specific membership query $MQ(q)$ are straightforward:

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

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

3.2 Prefix closure (P-filter)

If the language we want to learn consists of observations of runs of a real-world system, this language is obviously prefix closed, i.e., given a run, every prefix of this run is also in the language (as it is itself also a run of the system). This observation leads to a very powerful optimization, as the learning algorithm need not consider *continuations* of strings 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 counterexample, 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.

Definition 2 *A deterministic finite automata (DFA) is* prefix closed *if the set of nonaccepting states $S \backslash F$ are closed under the transition relation:*

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

Figure 1a shows a prefix closed DFA: the bottom state is the only nonaccepting state, and it is a sink. In this example, the transitions observed through testing are highlighted in black, and the additional transitions used to complete the DFA are colored gray. Note that any minimized prefix closed DFA has at most one nonaccepting state.

The prefix closure filter is also implemented by means of an optimized cache. The following two filter rules are used by the P-filter, whereby prefix(q) denotes the set of all prefixes of q:

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

3.3 Independence of actions (I-filter)

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 will result 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) reorderings. Our independence filter exploits this observation by only querying the system for *one member* of each such equivalence class. Whereas the prefix closure filter can always be employed, the independence filter requires the input of an application expert in the form of an independence relation that specifies which events can be *shuffled* in any order. As an example, the deterministic finite state machine in Fig. 1b contains the pair of independent actions (a, b), highlighted in black. Formally, independence is an irreflexive and symmetric relation on pairs of actions.

Definition 3 *Two actions $a, b \in \Sigma$ 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.*

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

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 that conforms 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 on a given ordering on the actions.

3.4 Symmetry (S-filter)

Hardware and telecommunication systems often contain large numbers of components that cannot be distinguished from each other by observation, i.e., without explicitly looking at their identification number. For example, from an observational point of view it often does not matter *which* device is performing a certain action (e.g., which memory bank is addressed or which phone calls a certain number); likewise the precise identification of the counterpart (the requesting processor or the receiver of the call) is not important 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 that grows with the number of identical components in a system. We implemented a corresponding filter that 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 at most as complicated as the real-world scenario with n actors (of a kind), where n is the maximum number of actors being active (not idle) in the model at the same time. Just as for independence of actions, it is the expert who determines which devices are considered equivalent in the sense above.

The implementation of the symmetry filter normalizes the queries as well. This is done by choosing a permutation that 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.

Figure 1c shows an example of DFA where (a, b) and the identity define a group of valid permutations.

4 Results: test-based model learning in practice

We have carried out model-construction experiments on several typical installations of the call-center application of [18]. For illustration purposes, we present four simple scenarios, each consisting of a telephone switch connected to a number of telephones (called "physical devices"). In each of these scenarios, the focus of the model was restricted to include a few actions of the telephones (inputs to the switch, formally denoted by A_I) and some responses of the switch (outputs, denoted by A_O). In the simplest scenario (S_1), just one phone is permitted to lift ($\uparrow$) and hang up ($\downarrow$) the receiver; in the last scenario (S_4), there are three phones where two (A and B) may establish a connection. For each of the first three, we also considered a variant scenario S_i' that includes an additional state-description output of the switch ($[hookswitch_D]$ for each device D).

S_1 One physical device (A),
$A_I = \{A \uparrow, A \downarrow\}$,
$A_O = \{initiated_A, cleared_A, [hookswitch_A]\}$.
S_2 Two physical devices (A, B),
$A_\text{I} = \{A \uparrow, A \downarrow, B \uparrow, B \downarrow\}$,
$A_\text{O} = \{initiated_{\{A,B\}}, cleared_{\{A,B\}}, [hookswitch_{\{A,B\}}]\}$.

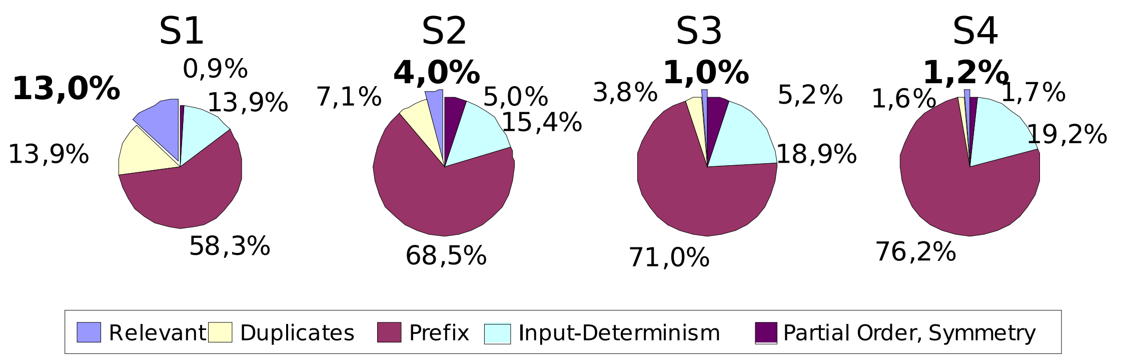


Fig. 2 Split of the membership queries by filter type for all scenarios

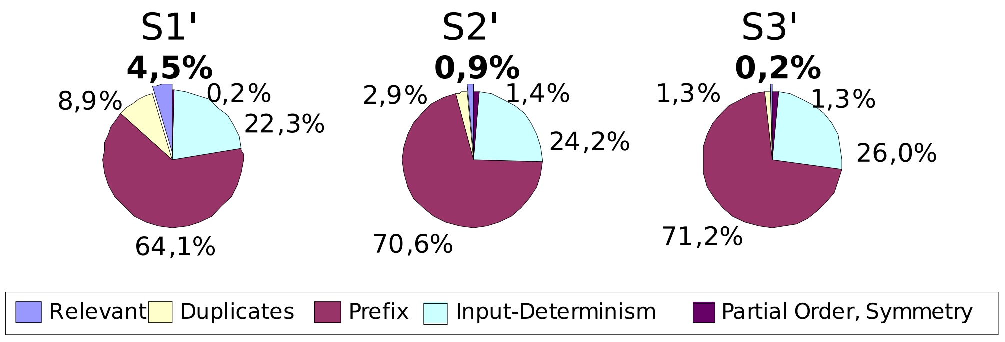


Fig. 3 Percentual filtering of membership queries per class of filter for the simplest scenarios

S_3 Three physical devices (A, B, C),
$A_I = \{A\uparrow, A\downarrow, B\uparrow, B\downarrow, C\uparrow, C\downarrow, \}$,
$A_O = \{initiated_{\{A,B,C\}}, cleared_{\{A,B,C\}},$
$[hookswitch_{\{A,B,C\}}]\}$.
S_4 Three physical devices (A, B, C),
$A_I = \{A\uparrow, A\downarrow, A \rightarrow B, B\uparrow, B\downarrow, C\uparrow, C\downarrow\}$,
$A_O = \{initiated_{\{A,B,C\}}, cleared_{\{A,B,C\}},$
$originated_A, established_B\}$.

In the following subsection, we discuss the impact of the filters to these scenarios.

4.1 Experimental analysis

Figure 2 lists in the second column the number of membership queries of the model relevant for the learning process and, in the last column, the number of membership queries L* needs to learn the model. Roughly speaking, the number of membership queries is polynomial (between quadratic and cubic) in the number of states. Columns 3–6 show the results of our optimized learning procedure, which exploits caching of posed queries (Duplicates column) as well as filtering according to the specific profile of our application scenario. Each of these columns lists the number of queries filtered out by the corresponding optimization when applied in the order given in the figure. As we see in the detailed diagrams of Figs. 3 and 4, a combination of input determinism of the systems, prefix closure of the language, and independence of certain actions allows one to significantly reduce the number of tests. In the most drastic case ($S3'$), we only needed a fraction of a percent of the number of membership queries required by basic L*. In fact, learning the corresponding automaton without any optimization would have taken about 4.5 months of computation time.

4.2 Discussion of results

The prefix reduction has a similar impact in all considered scenarios, as shown by the factors measured in Fig. 5, (column 5). This seems to indicate that it does not depend so much on the nature of the example and on its number of states.

The other two reductions (input determinism and partial order) vary much more in their effectiveness: the saving factor increases with the number of states. Shifting attention to the number of outputs and the lengths of output sequences between inputs seems to have a particularly high impact on the effects of the determinism filters. This can be seen by comparing the scenarios S_i with their counterparts S_i', which are much more laborious to learn. In these counterparts an additional output event is modeled, the hook-switch event, which occurs very frequently, that is, after each of the permitted inputs.

One would expect that the impact of the partial-order and symmetry impact would increase with the level of

Scenario	Relevant Querie	Duplicates	Prefix	Input-Determini	Partial Order, Sy	No Filter (pure L*)
S1	14	15	63	15	1	108
S1'	30	60	431	150	1	672
S2	97	173	1665	375	121	2431
S2'	144	453	10892	3725	211	15425
S3	206	745	13790	3674	1011	19426
S3'	288	1768	94198	34367	1719	132340
S4	1606	2161	100832	25456	2245	132300

Fig. 4 Percentual filtering of membership queries per class of filter for scenarios with state descriptor

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

Fig. 5 Reduction factors of membership queries by filter type

independence. Indeed this is confirmed by the experiments: S_1 has only one actor, so that there is no independence, which results in a factor of 1. As the number of independent devices increases, the saving factor increases as well, as shown by the figures for S_2 and S_3. The number of states does not seem to have any noticeable impact on the effectiveness of this filter, as the reduction factor more or less remains equal when switching from S_i to S'_i.

Compared to S_3, the saving factor in S_4 decreases. This is due to the fact that the action that has been added in S_4 (the initiation of a call) can establish dependencies between two devices, which reduces the partial-order and symmetry optimization potential.

5 A systematic experimental setting: the filter bench

To analyze the impact of the filters, we applied them to our automata learning algorithm of [15] and simulated the model generation process in a number of different scenarios. In what follows, we summarize how we created the target automata to be learned and how we set up the experiment with the different filters and filter compositions, and then we analyze the results.

5.1 Synthesizing target automata

We built a generator for automata with a realistic distribution of filter-relevant characteristics in order to have a sufficient number of target systems. These automata were prefix closed DFAs with some independent actions and some symmetries.

The automata generator works by first generating prefix closed DFAs in a random fashion, then taking some of them in parallel, thereby using the same notion of parallel composition as is typically used in the areas of hardware verification and testing, as well as for protocol specification and verification. This guarantees nontrivial independence and symmetry properties.

In the remainder of this section, we consider the scenario presented in [12]. We discuss in detail the measures on a DFA kernel automaton with four actions and seven states, which is taken three times in parallel, whereby the kernel DFAs synchronize on two actions. This way we generated 1000 prefix closed minimal DFAs, which arose from using 1000 randomly chosen kernel automata, with an average of 57 states, 8 input letters, 12 pairs of independent actions, and 6 valid symmetry reorderings. The presented experimental data represent the average measured for these 1000 target automata.

5.2 The filter bench

To analyze the efficiency of the optimization filters under equal experimental conditions, we ran all ten meaningful combinations of filters in parallel on the same outputs (membership queries) produced by the learning algorithm, thus allowing a direct comparison of their filtering power. The architecture of this analysis setting is depicted in Fig. 6.

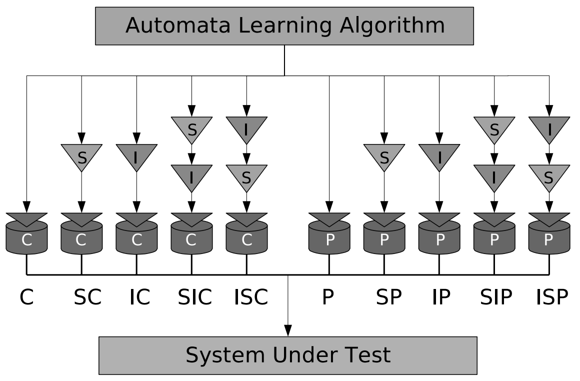


Fig. 6 The filter bench: filter compositions for efficiency and interference analysis

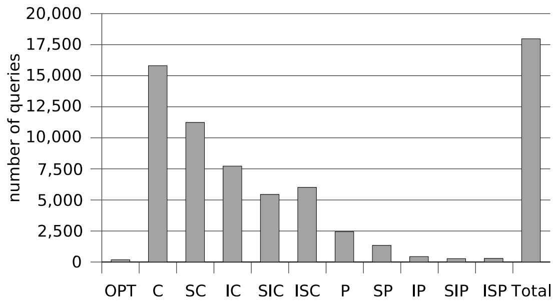


Fig. 7 Comparative filter efficiency by number of test cases

The simple caching for filtering redundancies (C) and the prefix closure filter (P) store information about queries. Thus it makes sense to compare their efficiency by running two distinct batteries of filter compositions, those using P-filtering, shown in the right-hand side of Fig. 6, and those using simple caching, shown on the left-hand side. We use the small database symbol to indicate that these filters store information and therefore require a nontrivial amount of memory.

In contrast, the symmetry (S) and the independence (I) filters only perform online reorderings of action; thus they do not need any extra memory. They are represented as small triangles.

Each filter composition is named after the initials of the filters it contains, in their order of application.

It is easy to see that several filter compositions do not make sense: one uses either redundancy caching or the stronger prefix closure, and since the reorderings lead to a higher probability of hitting cached test cases, the caching filters should be applied at the end of a composition. This leaves us with the ten filter combinations shown in Fig. 6.

6 Analysis of relative filter efficiency

Using the filter bench of Fig. 6, we measured for each of the ten filter compositions their filtering efficiency: given a target automaton to be efficiently learned, how many membership queries generated by L* (i.e., test cases for the target automaton) are recognized as irrelevant and thus filtered out by that filter composition?

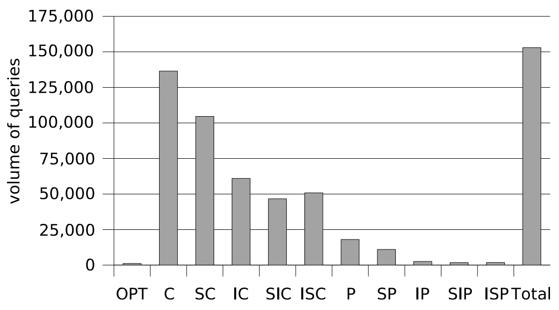


Fig. 8 Filter efficiency by volume of test suite

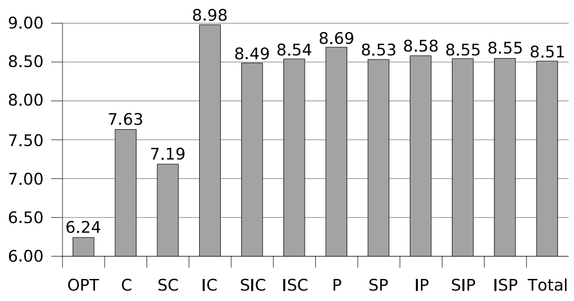


Fig. 9 Average test length

The results are summarized in Figs. 7 and 8. Two measures characterize the power of a filter composition: *how many* test cases remain and *which* testing effort remains.

6.1 How many test cases?

Figure 7 compares the test suite generated by pure L* (Total column) with the test suites arising after filtering in terms of number of test cases contained in the test suites. The optimal effort, achieved when all filter combinations are queried in parallel, is indicated in the column OPT.

We immediately see that the fraction of essential test cases is extremely low: of the 17.973 test cases requested by the plain L* application, corresponding to a volume of 153.001 stimuli, only 188 test cases with a total of 1171 stimuli are essential (see Table 1, rows Total vs. 13)! This confirms that optimization by filtering is an extremely powerful booster for the practicality of these techniques.

We also see that the consideration of prefix closure is extremely powerful in eliminating redundant test cases: all the compositions containing the P-filter perform much better than the corresponding C-based composition. In particular, the combination of the symmetry, independence, and prefix closure filters leads to a reduction in the learning effort close to the optimum.

Additionally, we observe that permutations of the normalizing filters do have effects on the efficiency of a filter composition: the SIC filter is more efficient than the ISC filter, and looking at the measured data, a similar ordering is measured

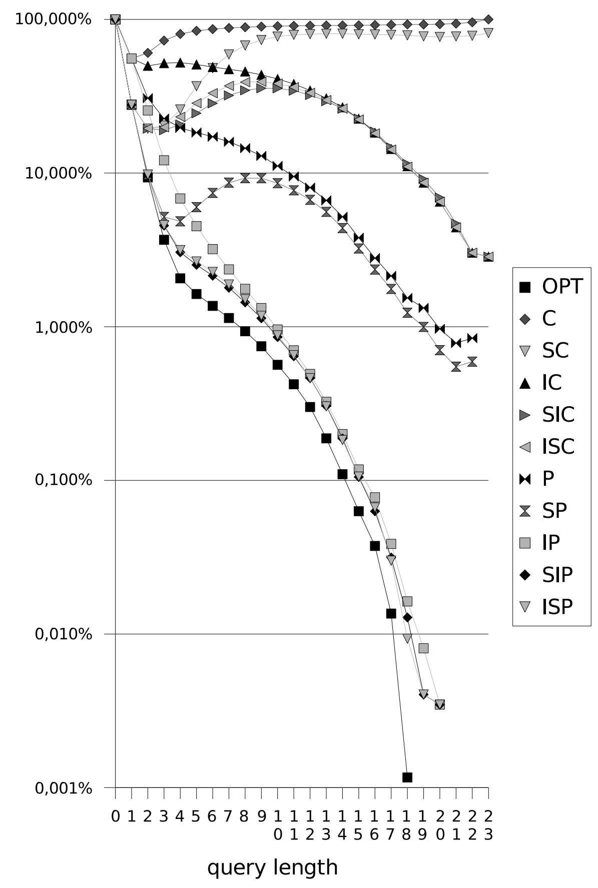


Fig. 10 Split of filter efficiency by test case length (logarithmic view)

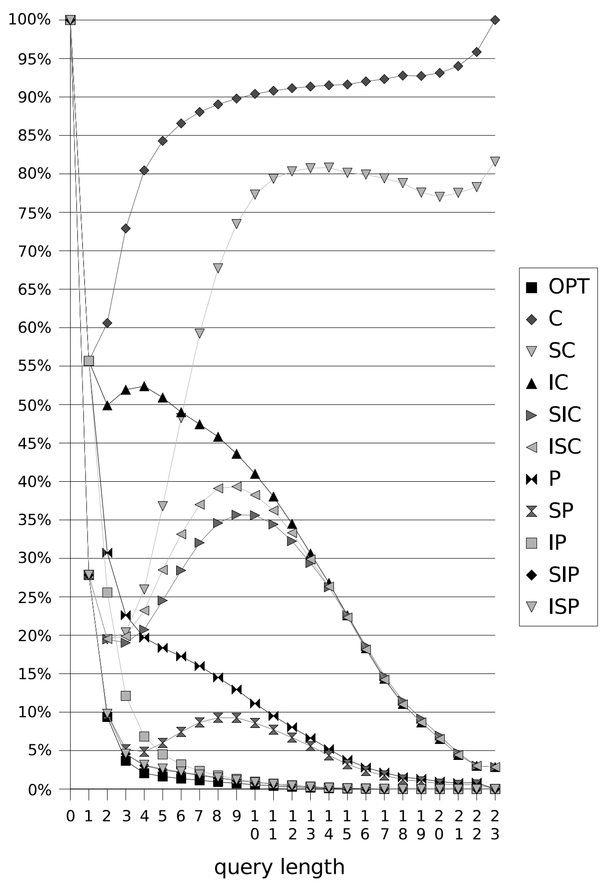


Fig. 11 Split of filter efficiency by test case length (linear view)

also for the SIP vs. ISP direct comparison, although it is of lesser importance. Thus we conclude that there is interference between the filters resulting in *second-order effects.*

6.2 Which test cases?

In order to more faithfully model the real effort of the learning procedure, we measured for each filter composition the number of required test stimuli of the resulting test suite. Figure 8 shows that of course the volume of the test suite is strongly related to the number of test cases, but as we see in Fig. 9, the average test length for each filter composition (computed as number of test stimuli/number of test cases) varies.

This led us to a more accurate analysis of this behavior by breaking down the analysis of the testing effort according to the length of the queries. Figures 10 and 11 provide a spectrum of the efficiency of the filter compositions in relation to the query length, in a logrithmic and a linear scale, respectively. Here we observe, e.g., that among the queries of length nine, corresponding to test cases with nine stimuli, 90% can be eliminated by the SIP filter composition, almost 74% can be eliminated by the SC composition, while the IC, ISC, and SIC can eliminate only between 45 and 35% of them. Pure prefix closure can eliminate about 15% of them, SP about 9%, and the remaining filter combinations are almost ineffective for queries of this length.

We also see that the relevant queries, the OPT curve, are typically very short, which gives us a good indication that effective learning of practical systems is realistic.

For the redundancy cache-based filters the situation is different: the simple cache filter C and the SC- combination work well for short queries, but not for long ones. But all cache-based filter combinations that include the independence filter are also good in the case of long queries.

7 Analyzing second-order effects in filter compositions

So far we have looked at each filter composition individually, but our goal is also to analyze the overlaps (due to dependencies and interferences) among filter compositions. This allows us to characterize the selectivity of filter combinations, i.e., which subset of filter compositions is able to filter out which queries. Since we have ten meaningful filter compositions C, SC, IC, SIC, ISC, P, SP, IP, SIP, ISP, there are 2^{10} possible partitions of the queries according to this filter selectivity criterion.

Indeed, during the learning process of all 1000 randomly chosen DFA Models, each single query generated by the plain

Table 1 Selectivity of filter compositions

Rank	Filter	Number of queries		Size of queries	
1	`0011111111`	3,809	21.2%	40,969	26.8%
2	`0000011111`	4,311	24.0%	37,320	24.4%
3	`1111111111`	2,161	12.0%	16,496	10.8%
4	`0111111111`	1,878	10.5%	15,291	10.0%
5	`0101111111`	1,312	7.3%	7,611	5.0%
6	`0011100111`	849	4.7%	7,476	4.9%
	...	...		...	
13	`0000000000`	188	1.0%	1,171	0.8%
	...	...		...	
28	`0000000010`	11	0.1%	84	0.1%
	...	...		...	
52	`0000000001`	1	0.0%	5	0.0%
53	`0000001000`	1	0.0%	4	0.0%
	Other	1,193	6.6%	8,379	5.5%
	Total	17,973	100.0%	153,001	100.0%

learning algorithm L* was filtered in parallel in our filter bench and classified in one of these 1024 partitions according to the set of filter combinations that was able to eliminate that query. The average results of this analysis are shown in Table 1

The table contains a row for each of the $2^{10} = 1024$ possible partitions of the queries, a selection of which is shown in Table 1. The second column contains bit vectors that characterize the sets of filter combinations by saying which filters (in the order given in Fig. 6) belong to that set. The rows are sorted according to the share of queries filtered out by that set of filter composition (columns 5 and 6).

1. As we see in the first row, 21.2% of all the queries generated by L*, corresponding to 26.8% of the total test patterns, could be filtered by any filter besides C and SC.
2. The second efficient set of filter compositions (rank 2) contains only those with prefix closure. This means in particular that a simple prefix closure filter is sufficient to filter out 24.0% of the queries, saving 24.4% of the testing effort in terms of test patterns.
3. 12% (almost 1/8) are truly redundant queries: they are eliminated by any filter.
4. 10% cannot be caught by cache only, but they are eliminated by any other combination.
5. 7.3% cannot be caught by IC, but for example by P (which are quite different principles, so we see that there is a certain specific insensitivity of some filter compositions).
6. Almost 5% escape caching (C or P) plus symmetry: they are clearly independence related.
7. On rank 13 we find the *relevant queries*: those that cannot be filtered by any filter. So for this kind of DFA only 1.0% of the queries generated by L* were relevant to learn the automaton.
8. Particularly interesting are the sets of combined filters on rank 28, 52, and 53: these values mean that there are some queries that can only be filtered by the SIP, ISP, and SP filters, respectively.

8 Related work

The systematic treatment of complex *black box* or *legacy* systems has attracted increasing attention as most software systems mutate to legacy systems over time: time constraints and short update cycles make it impossible to keep specifications and implementations manually in line. Thus automated validation techniques for black box systems are required.

The most established area in this respect is that of black box protocol testing, where one assumes a given finite-state-machine specification of the intended behavior of a protocol and intends to derive a test suite that checks that an implementation *conforms* to such a specification. There are several so-called *conformance testing* techniques for automatically generating test suites that guarantee that an implementation under test (IUT) conforms to a specification under certain hypotheses [4, 8, 23, 25].

A more recent line of development concerns checking whether an IUT satisfies certain correctness properties in the absence of a model or specification. Particularly promising are the approaches that employ techniques of automata learning, or *regular inference* [9, 10, 16, 21].

The relationship between machine learning and conformance testing was observed by Lee and Yannakakis [17, p. 1118], who stated that Angluin's algorithm can be used for fault detection. They also suggested an interesting subject of study, the relationship between conformance testing without reset (surveyed in [17]) and corresponding work on machine learning by Rivest and Schapire [22].

Both conformance testing and regular inference address the need for automated validation methods for black box systems by aiming at identifying the model structure underlying a black box system on the basis of a limited set of observations. However, there is a significant difference with strong impact on practicality: whereas conformance testing *checks* for equivalence with a *given* conjecture model, regular extrapolation[1] addresses the corresponding *synthesis* problem by means of techniques adapted from automata learning. Thus regular extrapolation is far more expensive [2]. It is our goal to exploit any knowledge about the system, like architecture, input/output structure, and knowledge about the application for optimization, to pragmatically overcome the complexity problems [11].

9 Conclusion

We have presented a thorough experimental analysis of the impact of, and the second-order effects between, a number of structurally rather different optimizations of a basic learning method. This systematic experimental investigation was motivated by observations made earlier in the context of an industrial project. Our workbench for automated learning allowed us to easily change the parameters for the random generation of target automata to be learned, to configure

[1] We prefer the notion of regular extrapolation to regular inference. Please consider these terms as synonyms here.

different optimization scenarios, and to automatically collect the corresponding analysis results, which confirmed our expectations. In fact, even using random automata (which profit much less from the filter technique than the real systems we had previously investigated), we were able to rank the filter combinations that reliably provide maximal test suite reduction, and thus a maximal learning speedup. Our results indicate that the practical use of automata learning is coming within reach.

Currently, we are experimenting with further optimizations. In particular we are starting a similar investigation for a variant of automata learning based on Mealy machines. We expect that the optimized Mealy scenario will give us a gain of another order of magnitude in learning speedup.

References

1. Angluin D (1987) Learning regular sets from queries and counterexamples. Inf Comput 2(75):87–106
2. Berg T, Grinchtein O, Jonsson B, Leucker M, Raffelt H, Steffen B (2005) On the correspondence between conformance testing and regular inference. In: Proceedings of FASE 2005, 8th international conference on fundamental approaches to software engineering. Edinburgh, UK, April 2005. Lecture notes in computer science, vol 3442. Springer, Berlin Heidelberg New York, pp 175–189
3. Brun Y, Ernst MD (2004) 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
4. Chow TS (1978) Testing software design modeled by finite-state machines. IEEE Trans Softw Eng 4(3):178–187
5. Cook JE, Du Z, Liu C, Wolf AL (2002) Discovering models of behavior for concurrent systems. Technical report, Dept of Computer Science, New Mexico State University, Las Cruces, NM
6. Cook JE, Wolf AL (1998) Discovering models of software processes from event-based data. ACM Trans Softw Eng Methodol pp 215–249
7. Ernst MD, Czeisler A, Griswold WG, Notkin D (2000) Quickly detecting relevant program invariants. In: Proceedings of the 22nd international conference on software engineering (ICSE 2000), pp 449–458
8. Fujiwara S, Bochmann Gv, Khendek F, Amalou M, Ghedamsi A (1991) Test selection based on finite state models. IEEE Trans Softw Eng 17(6):591–603
9. Groce A, Peled D, Yannakakis M (2002) Adaptive model checking. In: Proceedings of the 8th conference on tools and algorithms for the construction and analysis of systems (TACAS 2002). Lecture notes in computer science, vol 2280. Springer, Berlin Heidelberg New York, pp 357–370
10. Hagerer A, Hungar H, Niese O, Steffen B (2002) Model generation by moderated regular extrapolation. In: Proceedings of the 5th international conference on fundamental approaches to software engineering (FASE 2002). Lecture notes in computer science, vol 2306. Springer, Berlin Heidelberg New York, pp 80–95
11. Hungar H, Steffen B (2004) Behavior-based model construction. In: Int J Softw Tools Technol Transfer 6(1):4–14
12. Margaria T, Raffelt H, Steffen, B (2005) Analyzing second-order effects between optimizations for system-level test-based model generation. In: Proceedings of the IEEE international test conference (ITC), 8–10 November 2005. IEEE Press, Austin, TX
13. Niese O, Margaria T, Hagerer A, Brune G, Goerigk W, Ide H-D, Steffen B (2001) Automated regression testing of CTI-Systems. In: Proceedings of the IEEE European test workshop (ETW2001), May 2001, Stockholm, Sweden
14. 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. In: Annual review of communication, vol 55. International Engineering Consortium (IEC)
15. Hungar H, Margaria T, Steffen B (2003) Test-based model generation for legacy systems. In: IEEE international test conference (ITC), Charlotte, NC, 30 September–2 October 2003
16. Hungar H, Niese O, Steffen B (2003) Domain-specific optimization in automata learning. In: Proceedings of the 15th international conference on computer aided verification. Lecture notes in computer science, vol 2725. Springer, Berlin Heidelberg New York, pp 315–327
17. Lee D, Yannakakis M (1996) Principles and methods of testing finite state machines – a survey. Proc IEEE 84(8):1090–1126
18. Margaria T, Niese O, Steffen B, Erochok A (2002) System level testing of virtual switch (Re-)configuration over IP. In: Proceedings of the IEEE European test workshop. IEEE Press, Corfu, Greece
19. Mariani L, Pezzè M (2004) A technique for verifying component-based software. In: Proceedings of the international workshop on test and analysis of component based systems (TACOS 2004), Barcelona
20. Nimmer JW, Ernst MD (2002) Automatic generation of program specifications. In: Proceedings of the 2002 international symposium on software testing and analysis (ISSTA 2002), pp 232–242
21. Peled D, Vardi MY, Yannakakis M (1999) Black box checking formal methods for protocol engineering and distributed systems, (FORTE/PSTV). Kluwer, Dordrecht, pp 225–240
22. Rivest RL, Schapire RE (1993) Inference of finite automata using homing sequences. Inf Comput 103:299–347
23. Sabnani K, Dahbura A (1988) A protocol test generation procedure. Comput Netw ISDN Syst 15(4):285–297
24. Steffen B, Hungar H (2003) Behavior-based model construction. In: Mukhopadhyay S, Zuck L (eds) Proceedings of the 4th international conference on verification, model checking and abstract interpretation. Lecture notes in computer science, vol 2575. Springer, Berlin Heidelberg New York
25. Vuong ST, Chan WYL, Ito MR (1990) The UIOv-method for protocol test sequence generation. In: Proceedings of the 2nd international workshop on protocol test systems. North-Holland, Amsterdam, pp 161–176
26. Xie T, Notkin D (2003) Mutually enhancing test generation and specification inference. In: Proceedings of the 3rd international workshop on formal approaches to testing of software (FATES 2003) Lecture notes in computer science, vol 2931. Springer, Berlin Heidelberg New York, pp 60–69
27. Margaria T, Niese O, Raffelt H, Steffen B (2004) Efficient test-based model generationfor legacy reactive systems. In: Proceedings of international high level design validation and test workshop, Sonoma, CA (in press)