The Teachers' Crowd: The Impact of Distributed Oracles on Active Automata Learning *

Falk Howar[1], Oliver Bauer[1], Maik Merten[1], Bernhard Steffen[1], and Tiziana Margaria[2]

[1] Technical University Dortmund, Chair for Programming Systems, Dortmund, D-44227, Germany
{falk.howar|oliver.bauer|maik.merten|steffen}@cs.tu-dortmund.de
[2] University Potsdam, Chair for Service and Software Engineering, Potsdam, D-14482, Germany
margaria@cs.uni-potsdam.de

Abstract. In this paper we address the major bottleneck of active automata learning, the typically huge number of required tests, by investigating the impact of using a distributed testing environment (a crowd of teachers) to execute test cases (membership queries) in parallel. This kind of parallelization of automata learning has the best potential when the time for test case execution is dominant, an assumption valid for most practical applications. Our investigation explicitly focuses on the impact of the structure of the system under learning (number of states, size of alphabet) and the degree of supported parallelism. It comprises three variants of active learning algorithms with different test case generation profiles. These differences can be observed directly at the level of the run-times, which all show a linear speedup for moderate degrees of parallelization, but with different saturation points beyond which further parallelization does not pay off.

1 Introduction

Automata learning techniques are becoming a valuable tool in software construction. They have been used successfully to infer models of black-box systems in the context of model-checking [16], interface synthesis [1], and (model-based) testing [6]. In all these cases, models are a necessary prerequisite, and the lack of appropriate models can therefore be considered a show stopper. Active automata learning paves the way to overcome this problem by providing a solely test-based approach for inferring models of black-box systems. Key to this approach is the active interaction with the system under learning (SUL): The heart of the automata learning process consists of an elaborate way to generate test cases tailored to support the construction of adequate system models. This directly reveals the major bottleneck of the approach: active automata learning requires the execution of enormously many test cases. Already learning models

* This work is supported by the European FP 7 project CONNECT (IST 231167).

with a few hundred states typically requires the execution of some hundred thousand tests case on the SUL. In classical automata learning terminology, where the considered systems are simply deterministic automata, this means that huge numbers of so-called membership queries need to be answered by a (minimally adequate) teacher also called membership oracle [2].

In this paper we investigate the impact of using a distributed testing environment (a crowd of teachers) to execute test cases (membership queries) in parallel. In our experience, this kind of parallelization of automata learning has the best potential for speedup, as it allows for quite a flexible load balancing with very little overhead. Our investigation comprises three variants of active learning algorithms with different test case generation profiles that can be observed also at the level of parallelization-based speedup.

As conceptual basis we consider the sequential L^*_M algorithm presented in [19] that can be regarded as state of the art of practical automata learning. We present here its parallel version, which essentially arises from enabling parallel execution of test cases at three dedicated points. We have implemented this version together with two similar versions for learning algorithms with slightly different querying profiles in LearnLib[3], our library for experimenting with active automata learning algorithms [17, 13].

The paper presents an evaluation of these three parallel implementations in a series of experiments that explicitly focus on the impact of the structure of the SUL (number of states, size of alphabet) and the degree of supported parallelism in a context where the time for test case execution is dominant. This assumption is valid for most practical applications. The results indicate that for moderate degrees of parallelization a linear speedup can be realized, but that there seem to be saturation points beyond which further parallelization does not pay off.

Related Work. Active automata learning has been introduced in [2] for deterministic finite automata (DFAs). It has been extended to infer Mealy machine models in [15, 9], which are more adequate for describing reactive systems that produce outputs rather than accept or reject sequences of inputs. It has been used successfully to infer models of black-box systems in model-checking [16], testing [6], and interface synthesis [1].

The largest reported learned model concerns a software router with 22,000 states and 7 inputs [17]. Models of actual systems are easily bigger by some orders of magnitude. Thus, models inferred by active learning are valuable assets for documenting behavior of smaller systems or for organizing test suites. This is already useful in practice even if these models are not yet used in the verification of real systems.

The performance of active learning (in terms of queries) is essential when it comes to practical application. Performance has been addressed by applying application specific filters, which help answering membership queries without performing tests on a SUL [7]. The potential of different generic types of filters and the interplay of different filters has been investigated in [11, 10].

[3] http://www.learnlib.de

While optimizations to the number of membership queries have been investigated thoroughly, distributing queries has only been investigated from a theoretic perspective in [3]. In this paper, we investigate the impact of this optimization from a practical point of view.

Outline. We start in the next Section by providing some preliminary concepts and notation. In Section 3 we present our main result: an active automata learning algorithm that executes tests on a SUL in parallel. Section 4 presents the results of a practical evaluation of our new algorithm and compares its profile w.r.t. distribution of membership queries with some other active learning algorithms from LearnLib. Finally, we conclude in Section 5.

2 Preliminaries

In this section we define some preliminary concepts. We describe Mealy machines as a modeling formalism for a system under learning (SUL) and introduce a semantic model for Mealy machines that closely resembles regular languages. This model allows for the formulation of a Myhill/Nerode-like theorem for Mealy machines, which serves as the conceptual backbone of our learning algorithm.

Let Σ be a finite set of inputs of some system, e.g., method calls to a software library. We describe such system as an automaton.

Definition 1. *A* Mealy machine *is a tuple* $\mathcal{M} = \langle Q, q_0, \Sigma, \Omega, \delta, \lambda \rangle$ *where*

- Q *is a finite nonempty set of* states*,*
- $q_0 \in Q$ *is the* initial state*,*
- Σ *is a finite* input alphabet*,*
- Ω *is a finite* output alphabet*,*
- $\delta : Q \times \Sigma \to Q$ *is the* transition function*, and*
- $\lambda : Q \times \Sigma \to \Omega$ *is the* output function*.* □

A Mealy machine processes sequences of inputs (input words or simply words) and, other than a DFA, it produces outputs. We write $q \xrightarrow{a/o} q'$ to indicate that $\mathcal{M}$ moves from q to q' on input a producing output o according to δ and λ.

Abstracting from the fact that a Mealy machine will produce an output in every single step, we can describe the semantics of a Mealy machine as a function from the set of words to the set of outputs. Let ε denote the empty word, and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ be the set of all words of length greater than zero. Then, let $[\![\mathcal{M}]\!] : \Sigma^+ \to \Omega$ be the *set of traces* of $\mathcal{M}$.

In order to describe $[\![\mathcal{M}]\!]$ in terms of $\mathcal{M}$, we inductively extend the transition function δ to $\delta^* : Q \times \Sigma^* \to Q$ by defining $\delta^*(q, \varepsilon) = q$ and $\delta^*(q, aw) = \delta^*(\delta(q, a), w)$ for $q \in Q$ and $aw \in \Sigma^+$ with $w \in \Sigma^*$.

Let $[\![\mathcal{M}]\!]$ now simply be defined to be the last output of $\mathcal{M}$ on a particular word, i.e.,

$$[\![\mathcal{M}]\!](wa) \;=\; \lambda(\; \delta^*(q_0, w),\; a) \quad \text{for } wa \in \Sigma^+.$$

From this definition, we can immediately derive an equivalence relation of the set of words that resembles the Nerode relation for regular languages [14].

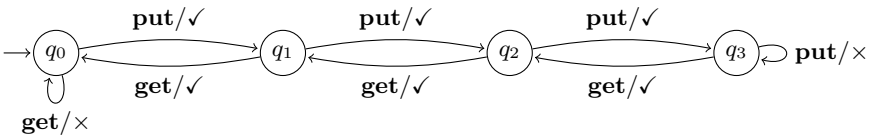

Fig. 1. Mealy machine model of a 3-place buffer

Definition 2. *For a mapping* $T: \Sigma^+ \to \Omega$*, two words* $u, u' \in \Sigma^*$ *are equivalent wrt.* T*, denoted by* $u \equiv_T u'$*, iff*

$$\forall v \in \Sigma^+ \,.\, T(uv) \;=\; T(u'v). \qquad \square$$

As for regular languages and DFAs, this relation can be used as a basis for the following characterization theorem, which we proved in [19].

Theorem 1. *A mapping* $T: \Sigma^+ \to \Omega$ *is a set of traces for some Mealy machine iff* $\equiv_T$ *has finite index.* $\square$

The theorem is a direct reformulation for Mealy machines of the well-known Myhill/Nerode theorem for DFAs. One direction of the proof of this theorem comprises the construction of the canonical Mealy machine $\mathcal{M}_T$ for T, in which every class of $\equiv_T$ corresponds to exactly one state in $\mathcal{M}_T$. This approach to automata construction is the conceptual backbone of all active automata learning algorithms. A detailed discussion can be found in [19].

Example. Figure 1 shows a graphical representation of a Mealy machine modeling a 3-place buffer with two inputs **put** and **get**, that add and remove elements from the buffer. States in the model correspond to the number of elements in the buffer. Successful actions on the buffer are indicated by output ✓ in the figure. The only two unsuccessful operations (indicated by output ×) are adding an element to the already full buffer and retrieving an element from the empty buffer. We will pick up this example when discussing the learning algorithm in Section 3. $\square$

3 Parallel learning

In this section we present our learning algorithm that distributes test cases to multiple systems under learning SULs, and a description of how we realized the distributed implementations of the active learning algorithms within LearnLib. The results from an experimental evaluation of the gained speedup are presented in the next section.

Active learning algorithms are formulated in the MAT-learning model [2], which assumes the existence of a *Minimally Adequate Teacher* (MAT) that answers two kinds of queries.

Membership queries test for the output of a word $w \in \Sigma^+$. Sometimes $MQ(w)$ will be used instead of $[\![SUL]\!](w)$ to emphasize that the value has to be determined by a test on the SUL.

Equivalence queries test whether an intermediate hypothesis automaton $\mathcal{H}$ is equivalent to the SUL, i.e., if $[\![\mathcal{H}]\!] = [\![SUL]\!]$. If hypothesis and system are not equivalent, an equivalence query delivers a *counterexample*, i.e., a word $w \in \Sigma^+$ for which $[\![\mathcal{H}]\!](w) \neq [\![SUL]\!](w)$. Equivalence queries will be denoted by $EQ(\mathcal{H})$ in pseudo-code listings.

Corresponding to the two kinds of queries, inference is organized in two phases, alternated iteratively. In the *hypothesis construction* phase a hypothesis model is derived from the observations and iteratively refined using membership queries. In the *hypothesis validation* phase hypothesis models are tested for equivalence with the SUL by means of equivalence queries.

3.1 The L^*_M algorithm

We present here a variant of the *reduced observation table* algorithm presented by Rivest and Schapire for determined finite acceptors (DFAs) in [18]. Our variant, the L^*_M algorithm, infers Mealy machine models and is described in [19], that also presents an extended example. Here we restrict ourselves to a brief description of the realization of the two phases discussed above in the basic algorithm and focus rather on the distribution of membership queries.

Hypothesis construction. In its basic form, active learning starts with a hypothesis automaton with only one state and refines this automaton on the basis of membership queries until a consistent state-minimal deterministic hypothesis automaton can be constructed. Key to achieving this result is the dual characterization of states that is established in Definition 2 and Theorem 1:

- by a set $U \subset \Sigma^*$ of prefixes. L^*_M constructs such a set U, containing prefixes $u \in \Sigma^*$ reaching all states of the hypothesis automaton. This characterization of states is too fine, as different words $u_1, u_2 \in U$ may lead to the same state in the target system. Hence, L^*_M maintains a second set $U_s \subseteq U$ of *access sequences* for which it is guaranteed that during learning $u_1, u_2 \in U_s$ (and $u_1 \neq u_2$) definitely lead to different states in the SUL. Technically, the relation between U and U_s is $U = U_s \cup U_s \times \Sigma$.
- by an ordered set, $V \subset \Sigma^*$, of *distinguishing sequences*. L^*_M realizes the characterization of a hypothetical state reached by some prefix u in terms of a vector $\langle r_1, \ldots, r_k \rangle$ (with $r_i \in \Omega$), characterizing the states by means of subsequent outputs, i.e., $r_i = [\![SUL]\!](u \cdot v_i)$ with $v_i \in V$. Intuitively, the vector approximates a characterization of the state wrt. to $\equiv_T$ (cf. Definition 2).

L^*_M maintains its observations in an *observation table* $\langle U, V, T \rangle$, where U is the set of prefixes, V is the set of suffixes, and $T : U \times V \to \Omega$ is the *table mapping* with $T[u, v] = [\![SUL]\!](u \cdot v)$. We define the *table row* of a prefix $u \in U$, denoted by $row(u)$, to be the vector $\langle T[u, v_1], \ldots, T[u, v_k] \rangle$ for $V = \langle v_1, \ldots, v_k \rangle$. Of course, membership queries are used to construct and maintain the table mapping.

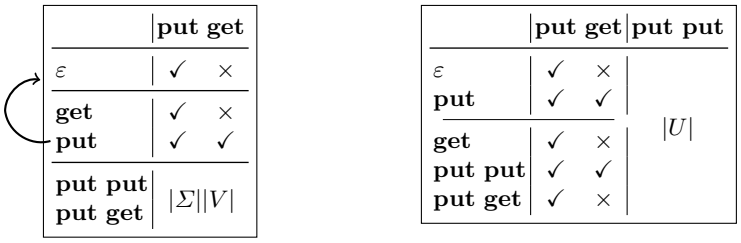

Fig. 2. Two intermediate observation tables with batches of membership queries for example from Figure 1.

Example. Figure 2 displays two snapshots of the observation table that is constructed when inferring a model of the 3-place buffer of Figure 1. The rows of the table are labeled with prefixes from U, the columns are labeled with suffixes from V, and the cells correspond to the table mapping; $row(u)$ is the actual row labeled by u in a table. The set U_s corresponds to the row labels in the upper parts of the tables. □

When learning, the set U_s is initialized as $\{\varepsilon\}$ and contains only the access sequence to the initial state. The set $U \setminus U_s$ is initialized accordingly as Σ and covers all transitions originating from the initial state. The ordered set V of distinguishing suffixes is initialized as Σ and characterizes states by the output that are produced along outgoing transitions.

The L_M^* algorithm then continues by refining the hypothesis. It checks whether an automaton constructed from the observation table is *closed* under the one-step transitions, i.e., if every transition from a state of this automaton ends in a well defined state of this same automaton. Since states in a hypothesis are characterized by rows in the observation table, this is the case if for every $u \in U \setminus U_s$ there exists a $u' \in U_s$ with $row(u) = row(u')$.

In case $row(u) \neq row(u')$ for some $u \in U \setminus U_s$ and all prefixes $u' \in U_s$, i.e., if the observation table is *unclosed*, the set of access sequences U_s is extended by u and U is extended by $\{u\} \cdot \Sigma$ accordingly. This procedure is iterated until the observation table is closed and a hypothesis automaton $\mathcal{H} = \langle Q, q_0, \Sigma, \Omega, \delta, \lambda \rangle$ can be constructed from the table, where:

- for every access sequence u in U_s there is a state $q_u \in Q$,
- $q_0 = q_\varepsilon$ is the state reached by the prefix ε, and
- for every prefix $ua \in U$ with $a \in \Sigma$ there is a transition $q_u \xrightarrow{a/T[u,a]} q_{u'}$ such that $row(ua) = row(u')$.

This construction resembles closely the construction of a canonical Mealy machine from a set of traces (cf. [19]). Closedness of the observation table guarantees that $\mathcal{H}$ is well-defined.

Hypothesis verification. Once a hypothesis $\mathcal{H}$ is produced from the observation table, an equivalence query can be used to find a counterexample, i.e., a word $w \in$

Algorithm 1 Parallel L_M^*

Input: A fixed set of inputs Σ
Output: A model $\mathcal{H}$ with $[\![\mathcal{H}]\!] = [\![SUL]\!]$

```
 1: U_s := {ε}                                          ▷ Initialize observation table
 2: U := U_s ∪ Σ
 3: V := Σ
 4: for all u ∈ U, v ∈ V do                             ▷ (|Σ| + 1)|V| membership queries
 5:     T[u, v] := MQ(uv)                               ▷ performed in parallel
 6: end for
 7: loop
 8:     while ⟨U, V, T⟩ not closed do
 9:         Let u in U \ U_s s.t. ∀u' ∈ U_s : row(u) ≠ row(u')   ▷ Close table
10:         U_s := U_s ∪ {u}
11:         U := U ∪ {ua | a ∈ Σ}
12:         for all a ∈ Σ, v ∈ V do                     ▷ |Σ||V| membership queries
13:             T[ua, v] := MQ(uav)                     ▷ performed in parallel
14:         end for
15:     end while
16:     construct hypothesis H from ⟨U, V, T⟩           ▷ cf. Section 3
17:     ce := EQ(H)                                     ▷ Perform equivalence query
18:     if ce = 'OK' then                               ▷ Learned successfully?
19:         return H
20:     end if
21:     decompose ce to find suffix v                   ▷ ⌈log2(|ce|)⌉ membership queries
22:     as described in Section 3                       ▷ performed sequentially
23:     V := V ∪ {v}
24:     for all u ∈ U do                                ▷ |U| membership queries
25:         T[u, v] := MQ(uv)                           ▷ performed in parallel
26:     end for
27: end loop
```

Σ^+ for which $[\![\mathcal{H}]\!](w) \neq [\![SUL]\!](w)$. In practice, an equivalence query is usually approximated by a number of membership queries, using ideas from conformance testing (e.g., [4]). Thus, equivalence queries can also be optimized by distributing membership queries to multiple instances of a SUL. This, however, exceeds the scope of this paper. Here we assume the existence of actual equivalence queries.

In [19] we proved the following theorem about counterexamples, which is a variant of the version for DFAs from [18].

Theorem 2. *A counterexample w has a suffix v such that for two prefixes $u \in U$ and $u' \in U_s$ with $u \neq u'$ and $row(u) = row(u')$ it holds that $[\![SUL]\!](u \cdot v) \neq [\![SUL]\!](u' \cdot v)$.* □

Intuitively, the theorem states that there is at least one suffix of the counterexample that will lead to unclosedness in the observation table if added to the set of distinguishing suffixes. Such a suffix can be found using a binary search on the counterexample [18, 19].

Adding the suffix to the set V leads to a refined hypothesis automaton in the next equivalence query, with at least one new state. Since, on the other hand, the characterization of states by rows in the table will never refine $\equiv_T$ (which would correspond to using Σ^+ as set of suffixes), the process is guaranteed to terminate with the canonical Mealy machine for SUL.

Put together, this results in Algorithm 1, where lines 1-6 initialize the observation table and lines 8-15 close the observation table as described above. In lines 16-17 an equivalence is used to search for a counterexample. In case none is found, the algorithm terminates successfully with the correct model in line 19. Otherwise, a new suffix is extracted from the counterexample and added to the table in lines 21-26.

3.2 Parallel execution of queries

In Algorithm 1 the parts between **for all** *domain* **do** and **end for** are meant to be parallelized, where *domain* defines a set of objects to be used in the different threads of execution. In particular, the algorithm performs *batches* of membership queries in parallel

- when initializing the observation table (lines 4-6),
- when closing the observation table (lines 12-14), and
- when adding new suffixes from counterexamples (lines 24-26).

The membership queries used in the analysis of counterexamples (lines 21-22), on the other hand, cannot be executed in parallel since the binary search on the counterexample cannot be split into multiple independent threads of execution.

Example. Figure 2 shows operations on the observation table for our running example. In the left table, the prefix **put** is added to the set of access sequences (indicated by an arrow). The resulting extension of the table is shown below: the two extensions of **put** with **put** and **get** are added to the set of prefixes. The resulting batch of membership queries has size 4 in this particular case. In the right table, the set of suffixes is extended by the new suffix **put put**, which will eventually lead to $row(\textbf{put}) \neq row(\textbf{put put})$. The corresponding batch of membership queries has size 5. □

3.3 Implementation

We have implemented a method for distributing membership queries on top of LearnLib. The conceptual idea of our realization is shown in Figure 3: The learning algorithm produces sets of membership queries (batches), which are handed to a so-called `BatchOracle`. This oracle schedules the distribution of the queries to a fixed number of `MembershipOracle`s, each of which is connected to a separate instance of the SUL.

Since this pattern is quite universal, we were able to extend all learning algorithms in LearnLib to support `BatchOracle`s. The next section discusses

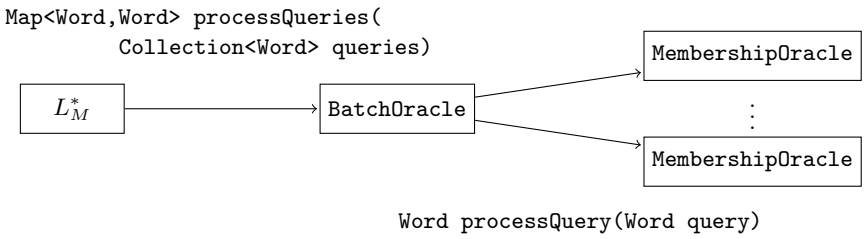

Fig. 3. Schematic overview of learning algorithm using multiple SULs in LearnLib

our experimental results showing that the different learning algorithms have different parallelization profiles.

4 Evaluation

In this section we present the results of three series of learning experiments conducted using the newly implemented parallelized versions of the learning algorithms in LearnLib. The experiments were executed in a simulation environment that supports equivalence queries. In order to compare the algorithms w.r.t. the distribution of membership queries, we voluntarily did not use membership queries to approximate equivalence queries. For the same reason we excluded membership queries used during analysis of counterexamples from the statistics since these (few) sequentially produced membership queries occur in all algorithms. Finally, we used a cache and counted only original membership queries. In the experiments in which we measured run-times, however, the time spent for analysis of counterexamples is included in the observations.

Our evaluation focuses on the impact of the structure of the SUL (number of states, size of alphabet) and the degree of supported parallelism in a context where the time for test case execution is dominant and thus the bottleneck, an assumption valid for most practical applications. In order to model this dominance without imposing too long experimentation times, we set up the simulator to require 5 ms (realized as busy waiting) per membership query. Although 5 ms are still extremely short when considering practical applications, where several seconds are not uncommon, they were sufficient to make the time spent in processing membership queries the dominant costs during learning while still allowing to execute experiments on systems of reasonable size in an acceptable amount of time. We therefore believe that the observed speedup patterns are quite representative, and that run-times for real systems of similar size can be easily extrapolated.

In the experiments we used the parallelized versions of the following three active learning algorithms:

$\mathbf{L^*_M}$ This is the algorithm we discussed in the previous section. It uses an observation table as underlying data structure, and it can produce batches of

queries when (1) resolving an unclosedness or when (2) extending the set of suffixes as discussed in Section 3.

DHC The *Direct Hypothesis Construction* (DHC) algorithm [19, 12] has a different profile w.r.t. to the batches of queries produced during learning. It is capable of resolving multiple occurrences of unclosedness at the same time, resulting in fewer but larger batches and a less uniform batch size. The implemented behavior would correspond to executing lines 8-15 of Algorithm 1 in parallel. When extending the set of suffixes the DHC algorithm behaves exactly like the L_M^* algorithm in lines 23-26.

Observation Pack Finally, the *Observation Pack* algorithm [5] uses a discrimination tree (cf. [8]) instead of an observation table. In this algorithm new prefixes are not added to the table but rather sunken into a tree. At every inner node of the tree a membership query is performed for each prefix. This leads to smaller batches than in the L_M^* algorithm: after resolving an unclosedness, k new prefixes can be sunken into the tree in parallel, resulting in batches of size at most k (the tree does not have to be balanced).
Also this algorithm uses new suffixes from counterexamples only to split one leaf of the tree (containing the set of prefixes corresponding to the incoming transitions of the hypothetical state represented at that particular leaf.) Thus, also batches produced by new suffixes tend to be smaller than in the L_M^* and the DHC algorithm.

Additionally, all three algorithms differ in the absolute number of membership queries and equivalence queries consumed during learning: While the Observation Pack algorithm uses dramatically less membership queries than the other two algorithms it uses, on the other hand, many more equivalence queries.

Experimental Setup We conducted three series of experiments, targeted at determining the profile of these algorithms along different dimensions of variability. All the experiments were conducted using LearnLib in Potsdam, on the Automata Learning server, a 2.4GHz AMD Opteron processor with 16 cores (= 16 threads) and 256GB memory running Linux. The first two series address profile trait inherent to the algorithm-specific organization of the learning process (in Sect. 4.1) and to the behaviour's scalability on a family of examples (in Sect. 4.2) and were performed in sequential learning mode. On the contrary the third series (in Sect. 4.3) concerns speedup and makes use of the multicore capability of the server.

4.1 Different profiles by example

In this first series of experiments we compared the three algorithms on identical SULs and analyzed in detail the resulting profiles in terms of the number of membership queries, the number of equivalence queries, the number of batches, and size of batches. Figure 4 explicitly summarizes the resulting batch sizes for each of the considered algorithms for one particular (small) SUL with 15 states and 5 inputs. Considering our other experiments, the data displayed in the figure are, however, quite representative.

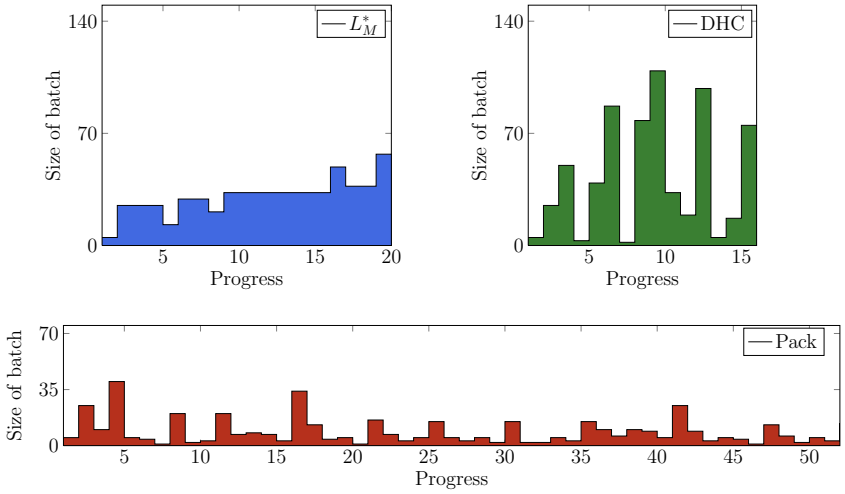

Fig. 4. Concrete numbers and sizes of batches for all algorithms and one SUL

1. The L_M^* algorithm produces batches of the same size during a single phase of hypothesis construction. Batches produced when adding new counterexamples tend to be bigger in size (especially later in the progress) than batches produced by unclosedness, which grow slowly and strictly monotonically with every new suffix.
2. The DHC algorithm, on the other hand, produces fewer but large batches of queries. Each spike in Figure 4 represents one phase of model construction. In the first two batches of every spike a new suffix is added for the initial state and then to every prefix. Then, some bigger batches follow resolving multiple occurrences of unclosedness at the same time, before the phase ends much faster than in the L_M^* algorithm.
3. The Observation Pack algorithm produces significantly more and smaller batches (of sizes of ca. 5 ($= k$)) than the other algorithms. In the case of L_M^* batch sizes for most of the batches are slightly above 25 ($= k^2$).
   Finally, the Observation Pack algorithm needs less membership queries (450) than the L_M^* and DHC algorithms (624 and 650), while these need fewer equivalence queries (4 each) than the Observation Pack (10).

4.2 Asymptotic profiles

In a second series of experiments we analyzed the mean size of batches produced by the learning algorithms for particular classes of target systems. In the first sub-series we randomly created canonical Mealy machines with 2^i states ($1 \leq i \leq 8$), 4 inputs and 8 outputs. In each class we performed 20 experiments and

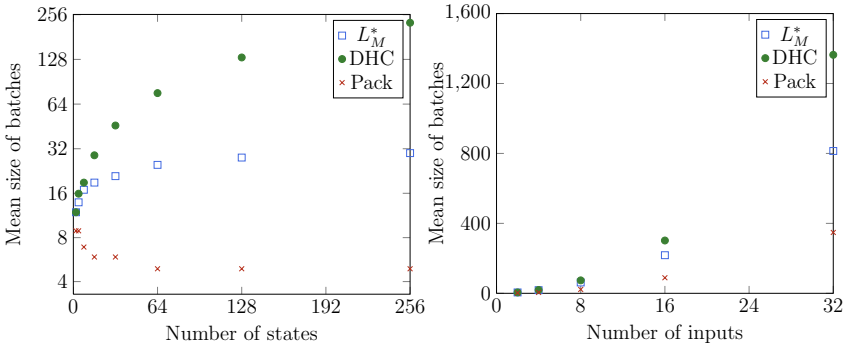

Fig. 5. Mean sizes of batches for SULs with increasing numbers of states (left) and for SULs with increasing numbers of inputs (right)

recorded the mean size of batches (mean of all batches of all experiments in a class).

The results are shown on the left of Figure 5. Please note that in this diagram the value axis is scaled logarithmically. The size of the batches produced by the DHC algorithm, develops almost linearly in the number of states, indicating that the number of occurrences of unclosedness that can be resolved in parallel grows with the size of the inferred automata. While the mean batch sizes for the L^*_M start from k^2 and increase moderately with the number of states (and suffixes needed to distinguish these), the mean batch sizes for the Observation Pack algorithm converge towards k, as was anticipated (see above). It may look surprising at first sight that the average batch size decreases with growing system size. This is simply due to the fact that the Observation Pack algorithm starts with a batch size of k^2, and that the influence of this initial batch size becomes less relevant the longer the learning process lasts.

In the second sub-series we fixed the number of states to 10, the number of outputs to 4, and varied the number of inputs in 2^i $(1 \leq i \leq 5)$. As in the first sub-series, we performed 20 experiments in every class and recorded the mean size of batches over all batches of all experiments in a class.

Figure 5 (right) shows the mean sizes of the created batches in the experiments. As is observed easily, increasing the number of inputs has (1) a more dramatic and (2) a more uniform effect on the mean sizes of batches for all learning algorithms: The L^*_M algorithm produces batch sizes very close to k^2, which is consistent with our expectations and with the discussion above. The behavior of the other algorithms does not meet their anticipated profile. This is due to the fact that in these experiments the number of inputs clearly dominates the number of states, leading to learning processes with only very few iterations of hypothesis construction and hypothesis validation. Thus the number of membership queries issued in the initial batch has a significant influence of the mean size of all batches.

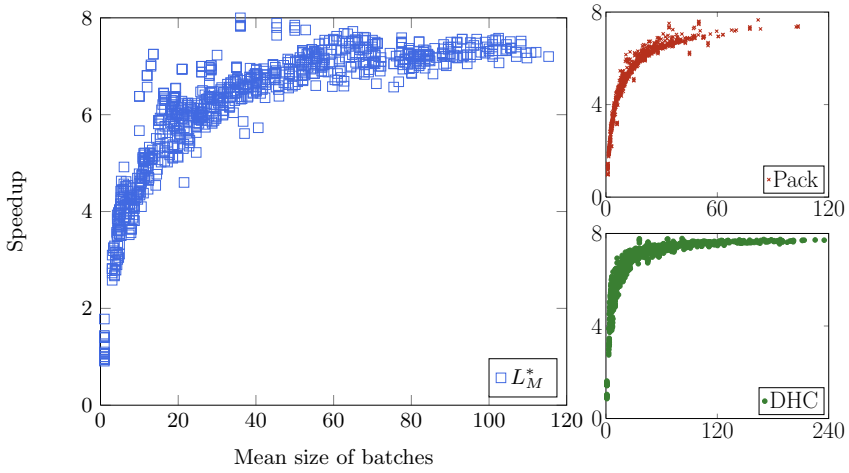

Fig. 6. Speedup per mean batch size for 8 SULs

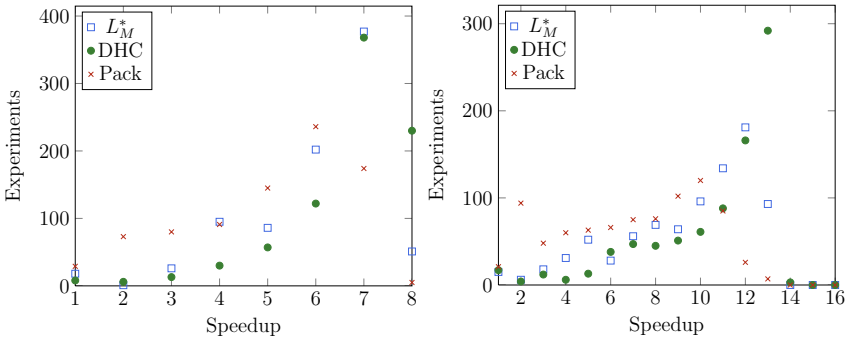

Fig. 7. Distribution of speedup in 850 experiments. Left: for 8 SULs. Right: for 16 SULs.

4.3 Speedup

We performed a third series of experiments in order to determine the speedup that can be gained for a fixed number (8 and 16 in this case) of SULs among which the membership queries can be distributed. For this series we have generated randomly a suite of 850 Mealy machine models, with varying number of states, number of inputs, and number of outputs. The constructed automata had up to 10 states, up to 20 inputs, and up to 10 outputs.

For every Mealy machine in this suite, we inferred a model three times: once using only one instance of the SUL to process membership queries (used as sequential control baseline), once using 8, and once using 16 instances of the SUL to process membership queries. Each SUL was allocated to a core (thread)

of the multicore server. We tried to use in different batches both the 8 and the 16 cores, in order to determine which grade of parallelism is most beneficial. As discussed above, we fixed the costs for a single membership query to 5 ms, which were spent in a busy waiting loop. The speedup in a single experiment is then the ratio between the runtime when using a single SUL and the runtime when using 8 (16) SULs. Figure 6 shows the speedup gained in the experiments with increasing mean size of batches (per experiment this time) in the case of 8 SULs.

Considering the results, two observations are striking:

- For eight SULs a speedup of nearly 8 can be actually realized in many cases.
- A speedup of 8 is not realized for mean batch sizes of 8 (or little above) but only for mean batch sizes greater than 40.

This phenomenon has to do with the quantization effect at the number of available cores. As soon as the batch size exceeds the number of cores, the job needs to schedule an additional cycle. The last such cycle is nearly always suboptimally used, as likely some cores are not needed and remain idle. For example, using 8 SULs, the worst case is a batch of size 9: needing more than 8 cores, it will require two 5ms cycles and 10 ms to compute. In the sequential case, 9 queries would require 45 ms. This results in a speedup of just 4.5. Growing batch size reduces the impact of not using all resources for the last queries of the batch, thus reaching better speedup. For example, a batch with 41 queries would compute with a speedup of 6.8 (205ms/30ms).

The results in Figure 6 show nicely the correspondence of mean batch sizes and speedup for the learning algorithms. The results are consistent for all three algorithms. The algorithms differ, however, in the distribution of achieved speedup on the set of 850 examples. This distribution is shown in Figure 7 for the case of 8 SULs (left) and 16 SULs (right) for all algorithms. In both series of experiments the Observation Pack algorithm achieves less speedup than the L^*_M algorithm, which in turn gains less speedup than the DHC algorithm. This is little surprising since the gained speedup correlates to the mean batch sizes.

Interestingly, none of the algorithms lead to a speedup of 14 or more in the experiments with 16 SULs. This can be attributed to the fact that for these examples no algorithm produced batch sizes large enough to fully leverage 16 SULs.

Summarizing the results, one can see that the potential speedup depends hugely on the system to be inferred (i.e., its size and the size of its input alphabet). Both factors have an influence on the mean batch size. To optimally leverage distribution of queries to multiple SULs, the expected mean batch size has to be considered (a) when choosing a particular learning algorithm and (b) when determining the number of SULs to be used.

Acknowledgement. We thank Mohamed Babiker for conducting preliminary experiments.

5 Conclusion

We have investigated the impact of parallelizing the execution of test cases (membership queries) during active automata learning in a context where the time for test case execution is dominant - an assumption valid for most practical applications. Our systematic investigation has focused on the impact of the structure of the SUL (number of states, size of alphabet) and the degree of supported parallelism. Our results indicate that for moderate degrees of parallelization a linear speedup can be realized, but that there seem to be saturation points beyond which further parallelization does not pay off.

Currently we are investigating how these results transfer to industrial-scale case studies, like e.g. the Springer's Online Conference Service (OCS), and how the different batch-profiles of the three considered algorithms show up there. Furthermore, we will investigate how this optimization can be combined with other optimizations - especially those that reduce the number of membership queries.

References

1. Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In *POPL'05*, pages 98–109. ACM, 2005.
2. Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.
3. José L. Balcázar, Josep Díaz, Ricard Gavaldà, and Osamu Watanabe. An optimal parallel algorithm for learning dfa. In *Proceedings of the seventh annual conference on Computational learning theory*, COLT '94, pages 208–217, New York, NY, USA, 1994. ACM.
4. Tsun S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, May 1978.
5. Falk Howar, Bernhard Steffen, and Maik Merten. From ZULU to RERS - Lessons Learned in the ZULU Challenge. In *ISoLA'10 (1)*, number 6415 in Lecture Notes in Computer Science, pages 687–704. Springer Verlag, 2010.
6. Hardi Hungar, Tiziana Margaria, and Bernhard Steffen. Test-based model generation for legacy systems. In *ITC'03*, pages 971–980. IEEE Computer Society, 2003.
7. Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-Specific Optimization in Automata Learning. In *CAV'03*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer Verlag, 2003.
8. Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA, 1994.
9. Tiziana Margaria, Oliver Niese, Harald Raffelt, and Bernhard Steffen. Efficient test-based model generation for legacy reactive systems. In *HLDVT'04*, pages 95–100. IEEE Computer Society, 2004.
10. Tiziana Margaria, Harald Raffelt, and Bernhard Steffen. Analyzing Second-Order Effects Between Optimizations for System-Level Test-Based Model Generation. In *ITC'05*. IEEE Computer Society, 2005.
11. Tiziana Margaria, Harald Raffelt, and Bernhard Steffen. Knowledge-based relevance filtering for efficient system-level test-based model generation. *Innovations in Systems and Software Engineering*, 1(2):147–156, July 2005.

12. Maik Merten, Falk Howar, Bernhard Steffen, and Tiziana Margaria. Automata Learning with on-the-Fly Direct Hypothesis Construction. *Submitted to ISoLA2011.*
13. Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next Generation LearnLib. In *TACAS'11*, volume 6605 of *Lecture Notes in Computer Science*, pages 220–223. Springer Verlag, 2011.
14. A. Nerode. Linear Automaton Transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.
15. Oliver Niese. *An Integrated Approach to Testing Complex Systems.* PhD thesis, University of Dortmund, Germany, 2003.
16. D. Peled, M. Y. Vardi, and M. Yannakakis. Black Box Checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225–246, 2002.
17. Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. LearnLib: a framework for extrapolating behavioral models. *Int. J. Softw. Tools Technol. Transf.*, 11(5):393–407, 2009.
18. Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.
19. Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to Active Automata Learning from a Practical Perspective. In *Formal Methods for Eternal Networked Software Systems, SFM'11*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer Verlag, 2011.