Master Thesis Computing Science

Radboud University Nijmegen
August, 2015

Performance improvement in automata learning

Speeding up LearnLib using parallelization and checkpointing

Author: Marco Henrix
First supervisor: Dr.ir. Jan Tretmans
Second supervisor: Dr. David Jansen
Third supervisor: Prof. dr. Frits Vaandrager

Contents

1 Introduction

Nowadays software can be found all around us. Not only on a typical server or desktop computer, but also embedded into washing machines, cars, airplanes, medical equipment and even light bulbs. Unfortunately, bugs are common in software. There are bugs that, for example, only introduce a minor glitch during playing a video game, but there are also bugs that can have catastrophic consequences. In 2003, a gigantic blackout that cut off electricity to 50 million people in the USA and Canada happened due to a software bug [16].

But even worse, software failure can sometimes be fatal. Recently an Airbus A400 plane crashed and caused the death of four crew members. This crash happened due to a software failure introduced during the installation of the flight control software [28]. Another recent example is the Boeing 787 Dreamliner. It turned out that a software bug causes total loss of power of the airplane if the system runs uninterrupted for 248 days [46]. Luckily, this bug has not yet led to any plane crash, but it could be imagined how catastrophic the consequences could be. Dozens of fatal car accidents with Toyotas were caused due to unintended acceleration. Toyota was fined for more than a billion dollar. The badly designed software that should control the engine probably played a role in these accidents [37].

We rely more and more on software, as we can see for example with the emergence of self-driving cars. Thus it becomes more and more important to prevent software failures.

Model-based testing is an effective approach to reduce the number of software failures and to improve software quality. A model, such as a finite state machine could be used to get more insight in how the software should work. The model could also be formally verified with tools like UPPAAL [30]. If there is for example a model of the software of the traffic lights at a road intersection, it can be proven that the lights of crossing directions can not be green at the same time. Using Model-based testing tools such as JTorX [31], test cases could be generated from the model, to test whether the actual implementation is conforming to the model.

A problem with model-based testing is how the model should be generated. Often this is done manually, which can be very time-consuming and error-prone. It would be much easier and it would make model-based testing more accessible if these models could be generated automatically. A model could be learned to verify properties of existing software. It also could be used to learn a model of a previous version of the software and test whether the new version has got any newly introduced faults (regression testing). It can be useful in the case of legacy software, where the source code has been lost or can hardly be understood due to the lack of documentation. But it is also useful for security analysis by which the learned model can be analysed for potential security vulnerabilities. The automatic learning of state machine models from an actual implementation is called automata learning.

There are two kinds of automata learning: passive and active. In passive automata learning there is no interaction between the learner and the system under learning (SUL); a model should thus be learned from existing log files. Contrary, in active automata learning there is interaction between the learner and the SUL. The learner sends inputs to the SUL and dependent on the output new inputs are generated. By applying this repeatedly, a model will be constructed. In this thesis we will focus on active automata learning. A popular tool to perform active automata learning is LearnLib. [41]

LearnLib has been successfully used to learn models of real world software implementations such as network protocols (TCP [36], SSH [45], SIP [23]), bank cards [21] and control software of printers [42].

A persistent problem of LearnLib is that the time required to learn a model can be very long. For example one study showed that the learning of a real world implementation of printer control software took more than 11 days [42]. The required time to learn a model will even greatly increase when the number of possible inputs of the SUL increases, the number of states in the SUL increases, it takes longer to reset the SUL or when it takes longer to process an input by the SUL. (In section 2.2 we give detailed information about the complexity of the algorithm.) To give an indication about this: if the execution of inputs during the learning of the printer control software [42] would be slower, instead of about 4000 inputs per second only 10 inputs could be executed in a second, which is a very plausible scenario (there are a lot of systems that interact slower), the execution would take about 400 times longer, which means a learning time of more than 12 years.

1.1 Research question

The research question to be answered in this thesis is:

How can the execution speed of LearnLib be improved?

For this the following sub questions will treated:

- What are possible improvement techniques?
- What are the theoretical effects of these techniques?
- What are the effects when these techniques are applied in practice? How big is the actual improvement?

1.2 Possible improvement techniques

We distinguish 4 different categories of improvement techniques:

- Modifying or adapting of the algorithm used by LearnLib to reduce the number of inputs that are sent to the SUL. Many other studies, such as [43] and [42], have focussed on these kinds of improvements. Many of these studies modify or replace the *equivalence oracle* (See section 2.2).
- By applying abstraction the number of different inputs and outputs can be reduced. Suppose we want to learn the model of a vending machine. This machine has buttons for every type of candy, so there is a Mars button, Snickers button, Twix button, etc. Using abstraction all these different inputs can be seen as a single 'candy button' input by the learning algorithm, but there should be a made a mapping so that during the execution of such an input on the actual SUL a specific button will be pressed. This way the model to be learned will be much smaller and easier to understand and the total number of inputs that are sent to the SUL by LearnLib will be reduced. This has been researched in previous studies such as [25] and [22]. During these studies the tool Tomte is developed, which is a tool that fully automatically constructs such abstractions for automata learning.
- Speeding up the internal code of LearnLib. For example it can be researched whether more efficient data structures could be used, to reduce

the overhead of LearnLib itself during the learning process. However, we want to focus during this research on relatively slow SULs (where processing a single input takes at least 10ms). In that case the current overhead of LearnLib seems negligible. Therefore we are not interested in researching such improvement techniques.

- More efficient execution of the inputs as they are generated by LearnLib. The algorithm as used by LearnLib to generate the inputs that should be executed by the SUL will not be modified, but by applying some practical 'tricks' the execution speed of LearnLib might be improved a lot. We have decided to focus on these kind of improvements in this thesis, since there have not been many studies yet that research these kinds of improvements and we have some interesting improvement ideas. In the following sections we describe the two improvement techniques that we are going to research in this thesis: parallelization and checkpointing. A third one is described in section 9.2, but will not be researched in this thesis.

1.3 Proposed improvement techniques

- Parallelization - The current typically used implementation of LearnLib is single threaded, which means that even if you would have access to a large computer cluster the required execution time will remain about the same. By executing multiple instances of the SUL at the same time the total learning time can be reduced. Thus the total amount of work that has to be done will not be reduced, but by distributing it, the work can be done in less time. This is challenging since it is not clear which parts actually can be parallelized and the scalability is also not known. If the performance increases a lot if there are 2 parallel instances used instead of 1, it does not automatically mean that the performance also can increase equally if there are used 20 parallel instances instead of 10.

- Saving and restoring software states (checkpointing) - This concept is based on the notion of the software state of the SUL. It could be only applied in active automata learning and not in passive automata learning, since the concept requires a running SUL of which the software state could be saved and reused at a later time during the learning process. This is challenging since such a technique has never been applied in this context before as far as we know. Implementing it is far from trivial, the methods which we are going to use in this thesis, such as DMTCP 5, only exist for a couple of years and are still experimental. It is also not trivial to decide which software states should be stored (See section 4.3).

  In the following section 1.4 this technique will be explained further.

- Combination of both of the above techniques. Both proposed techniques do not necessarily hinder each other, so they can be combined. This way even greater speed improvements might be possible.

1.4 Saving and restoring states

When LearnLib learns a model it sends queries to the SUL. A query consists of resetting the SUL to bring it into the initial state and thereafter executing a sequence of inputs. A typical fragment of the execution of the queries that are generated by LearnLib when learning a model of a coffee machine will look like:

```
...
Machine reset > coffee button > start button
Machine reset > coffee button > start button > start button
Machine reset > coffee button > start button > coffee button
...
```

All these queries share the same beginning that will be executed over and over again. To prevent spending time on duplication of work the following improvement is proposed:

```
...
Machine reset > coffee button > start button > save the state of the SUL
Restore the state of the SUL > start button
Restore the state of the SUL > coffee button
...
```

After the first occurrence of the shared beginning the state of the SUL will be saved. Each following time a query with the same beginning has to be executed, that beginning could be skipped. Instead the earlier saved state could be restored. A saved state is called a *checkpoint*. It would be nice if the SUL itself offers the possibility to save and restore its state, but in this

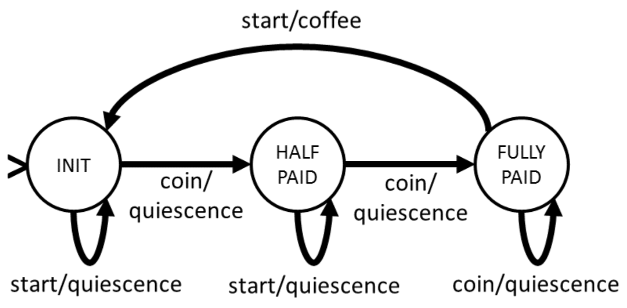


Figure 1: Example of a Mealy machine.

thesis we will focus on a generic solution that can also be used even if the SUL does not support saving and restoring its state by itself.

The concept of reusing the state of previously executed queries is not completely new. There has been one study that uses this concept [29], but there were some major drawbacks with their approach, such as that it was not a generic solution, as will be discussed further in section 4. Our approach using saving and restoring states of an arbitrary SUL is completely new as far as we know and does provide a generic solution.

2 Preliminaries

Before discussing possible optimizations, we will first briefly introduce the underlying technique on which LearnLib is based. For a more elaborative description, we refer to [40] and [44].

2.1 Mealy machines

The kind of state diagrams that LearnLib uses are Mealy machines. These are state diagrams that have a finite amount of states and that are deterministic. The transitions are specified by a tuple of an input and a corresponding output.
Figure 1 shows an example of a Mealy machine. This machine produces coffee after inserting at least 2 coins and pressing the start button. In all other cases this machine gives no output (quiescence).

Formally a Mealy machine is defined by a 6-tuple : $M = (I, O, Q, q_0, \delta, \lambda)$ where

- I is the finite alphabet of input symbols
- O is the finite alphabet of output symbols including quiescence
- Q is the finite, non-empty set of states
- q_0 is the initial state ($q_0 \in Q$)
- $\delta : Q \times I \to Q$ is the transition function
- $\lambda : Q \times I \to O$ is the output function

The formal definition of the Mealy machine as shown in Figure 1 would be:

- $I = \{$start,coin$\}$
- $O = \{$coffee,quiescence$\}$
- $Q = \{$init,half paid,fully paid$\}$
- $q_0 =$init
- δ(init,coin) =half paid; δ(init,start) =init;
  δ(half paid,coin) =fully paid; δ(half paid,start) =half paid;
  δ(fully paid,coin) =fully paid; δ(fully paid,start) =init
- λ(init,coin) =quiescence; λ(init,start) =quiescence;
  λ(half paid,coin) =quiescence; λ(half paid,start) =quiescence;
  λ(fully paid,coin) =quiescence; λ(fully paid,start) =coffee

2.2 Learning algorithm

In figure 2 we can see an overview how a learning experiment in LearnLib works. It consists of an experiment object that controls the learning, a so-called *membership oracle* that constructs a model, a so-called *equivalence oracle* that verifies whether a learned model is correct and a SUL on which the membership oracle and equivalence oracle can execute queries. To learn a model first a command is sent to start the learning on the membership oracle (1). Then a number of times the membership oracle sends queries (which

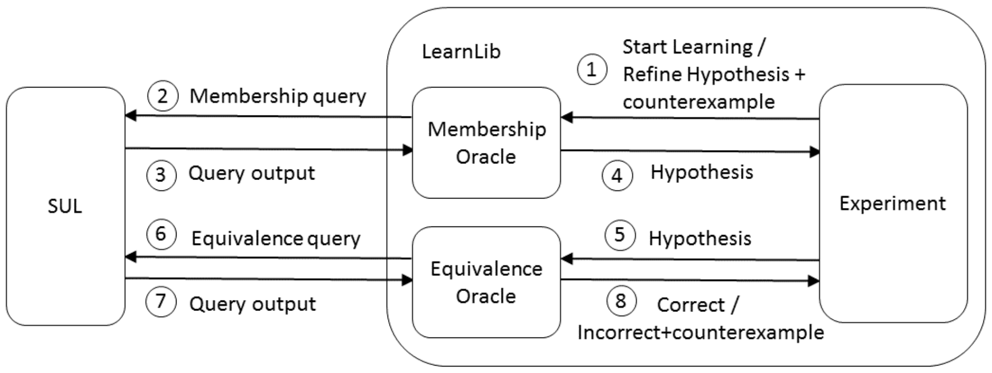


Figure 2: Abstract overview of learning a model with LearnLib

are called *membership queries*) to the SUL (2) and observes the output (3). Based on the outcomes of those queries a hypothesis (a model that might be correct) is constructed that is returned to the experiment object (4). Subsequently the hypothesis is passed to the equivalence oracle (5). To determine whether the hypothesis is correct the equivalence oracle can send multiple queries (which are called *equivalence queries*) to the SUL (this is called a *round*) (6) and observe the corresponding output (7). Next the equivalence oracle reports its finding whether the hypothesis is correct or not to the experiment object (8), in case the hypothesis is incorrect the equivalence oracle also provides a counterexample. If the hypothesis was found to be correct by the equivalence oracle, the algorithm will finish, otherwise it starts all over by sending a command to refine the hypothesis to the membership oracle accompanied by the found counterexample (1).

The algorithm that LearnLib uses to learn a state machine is based on Angluins L^* algorithm [26]. However the algorithm originally described by Angluin uses deterministic finite automata instead. In LearnLib an adapted version for Mealy machines of this algorithm is used. Note that the L^* algorithm does not define how the equivalence oracle should work.

The algorithm uses a so called *observation table* to keep track of the results of executed queries and to determine which queries are yet to be carried out. In table 1 we could see an example of such an observation table.

Formally, an observation table $\mathcal{OT}$ is defined by a tuple (S, E, T) where

- $S \subseteq I^*$ is a finite, prefix-closed set
- $E \subseteq I^*$ is a finite, suffix-closed set
- T is a finite mapping of strings of $(S \cup S \cdot I) \cdot E$ to elements from the output alphabet O

From this definition we can deduce $row(s)$. Let $s \in (S \cup S \cdot I)$, then $row(s)$ denotes the function f defined by $f(e) = T(s \cdot e)$. Now the following two properties can be defined:

- An observation table $\mathcal{OT}$ is called **closed** if $\forall t \in S \cdot I . \exists s \in S . row(t) = row(s)$
- An observation table $\mathcal{OT}$ is called **consistent** (or semantic suffix-closed) if $\forall s_1, s_2 \in S . row(s_1) = row(s_2) \Rightarrow \forall i \in I . row(s_1 \cdot i) = row(s_2 \cdot i)$

The algorithm terminates after at most $n^2k + k^2n + n \cdot log(m)$ membership queries, where n is the number of states in the SUL, k is the size of the set of inputs and m is the length of the longest counterexample.

When we apply the adapted L^* algorithm to the example from figure 1 we will get first the result shown in observation table 1. Note that this required the execution of six membership queries (namely: coin, start, coin · coin, coin · start, start · coin, start · start).
Since this observation table is closed and consistent. The first hypothesis will be generated, as shown in figure 3. But this first hypothesis is not correct, thus the equivalence oracle should provide a counterexample. Let's assume the given counterexample is: coin · coin · start
Then the L^* algorithm will add all the prefixes of the counterexample (coin, coin · coin and coin · coin · start) to S and will update the observation table. The result of these steps are shown in observation table2. Eight additional results are included in the table, of which two can be derived from the counterexample (namely: coin·coin and coin·coin·start), the remainder had to be found by executing membership queries.
Although the second observation table is closed, it is not consistent. Therefore L^* will expand E, in this case with: coin · start. The result is shown in observation table 3. A column with seven results is added. However if you

Algorithm 1 L^* adapted for Mealy machines. Algorithm from [40]

.

Input Alphabet A_I, Output Alphabet A_O, Observation Table $\mathcal{OT} = (S,\ E,\ T)$, where initially $S = \{\epsilon\}$ (the empty input string) and $E = I$.

```
do
    $\mathcal{OT} \leftarrow update(\mathcal{OT})$
    while $(\neg isClosed(\mathcal{OT}) \vee \neg isConsistent(\mathcal{OT}))$ do
        if $(\neg isClosed(\mathcal{OT}))$ then
            $\exists s_1 \in S,\ i \in I.\ \forall s \in S.\ row(s_1 \cdot i) \neq row(s)$
            $S \leftarrow S \cup \{s_1 \cdot i\}$
            $\mathcal{OT} \leftarrow update(\mathcal{OT})$
        end if
        if $(\neg isConsistent(\mathcal{OT}))$ then
            $\exists s_1,\ s_2 \in S,\ i \in I,\ e \in E.\ row(s_1) = row(s_2) \wedge$
            $T(s_1 \cdot i \cdot e) \neq T(s_2 \cdot i \cdot e)$
            $E \leftarrow E \cup \{i \cdot e\}$
            $\mathcal{OT} \leftarrow update(\mathcal{OT})$
        end if
    end while
    $M_c \leftarrow M(\mathcal{OT})$
    $\sigma_c \leftarrow EO(M_c)$
    if $(\sigma_c \neq \perp)$ then
        $S \leftarrow S \cup Prefix\ (\sigma_c)$
    end if
while $(\sigma_c \neq \perp)$
```

EO: Mealy machine $\rightarrow I^*$ denotes the call to the equivalence oracle.

The function *update* is defined as follows: *update* : $\mathcal{OT} \rightarrow \mathcal{OT}$
where for each $s \in (S \cup S \cdot I)$ and $e \in E$, $T(s \cdot e) = MO(s \cdot e)$
MO denotes the execution of a single membership query by the membership oracle

Prefix denotes the function that generates the set of all possible prefixes for a given input string of type I^*, including the full input string itself

The function M is used to compute the *hypothesis* out of the observation table as follows: $M : \mathcal{OT} \rightarrow$ Mealy machine
$Q = \{row(s) | s \in S\}$,
$\delta(row(s),\ a) = row(s \cdot a)(s \in S,\ a \in A_I)$,
$\lambda(row(s),\ a) = T(s \cdot a)(s \in S,\ a \in A_I)$,
$q_0 = row(\epsilon)$ (the initial row)

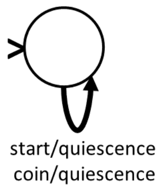


Figure 3: First hypothesis

take a close look, three of these results (namely for the queries coin·start, coin·coin·start and coin·coin·coin·start) were already present in observation table 2).
Observation table 3 is both closed and consistent, thus a second hypothesis will be derived from table 3. This hypothesis is identical to the one as shown in figure 1, no counterexample could be found by the equivalence oracle, so this hypothesis is correct.

$\mathcal{OT}$		E	
		coin	start
S	ϵ	quiescence	quiescence
$S \cdot I$	coin	quiescence	quiescence
	start	quiescence	quiescence

Table 1: First observation table

$\mathcal{OT}$		E	
		coin	start
S	ϵ	quiescence	quiescence
	coin	quiescence	quiescence
	coin· coin	quiescence	coffee
$S \cdot I$	start	quiescence	quiescence
	coin · start	quiescence	quiescence
	coin · coin · coin	quiescence	coffee
	coin · coin · start	quiescence	quiescence

Table 2: Second observation table

2.3 Equivalence oracle

In contrast to the membership oracle, the implementation of the equivalence oracle is far from trivial. Where the membership oracle simply has to ex-

$\mathcal{OT}$		E		
		coin	start	coin · start
S	ϵ	quiescence	quiescence	quiescence
	coin	quiescence	quiescence	coffee
	coin · coin	quiescence	coffee	coffee
$S \cdot I$	start	quiescence	quiescence	quiescence
	coin · start	quiescence	quiescence	coffee
	coin · coin · coin	quiescence	coffee	coffee
	coin · coin · start	quiescence	quiescence	quiescence

Table 3: Third observation table

ecute the provided query on the actual SUL and return the corresponding output, the equivalence query has to 'magically' determine whether a given hypothesis is correct. The originally described algorithm by Angluin didn't provide a method how to implement such an equivalence oracle. Luckily, several methods to implement an equivalence oracle have been developed later. In this thesis we will focus on the *random words* equivalence oracle, because it is easy to understand, easy to use and already implemented in LearnLib (in a non-parallelized version). A minimum query length (QL_{min}), maximum query length (QL_{max}) and a maximum number of queries have to be specified by the user. This algorithm randomly executes a random query on the SUL with a length $\geq QL_{min}$ and $\leq QL_{max}$. If the result of the executed equivalence query differs from what is expected from the hypothesis, the equivalence oracle immediately terminates and the query will be returned as a counterexample. Otherwise if the specified maximum number of queries has not been reached yet this procedure of executing a random query will be repeated.

3 Applying parallelism

Figure 4 shows the concept of parallelization of the learning process. There is still one instance of LearnLib, but multiple instances of the same SUL plus potential helper programs that are needed to run the SUL and to communicate with it.
Helper programs could provide a simulated environment. Such a simulated environment can simulate the physical environment the embedded software has to operate in, which can make testing more useful. For example in a self driving car the software could steer left to take a turn, but by simulating the

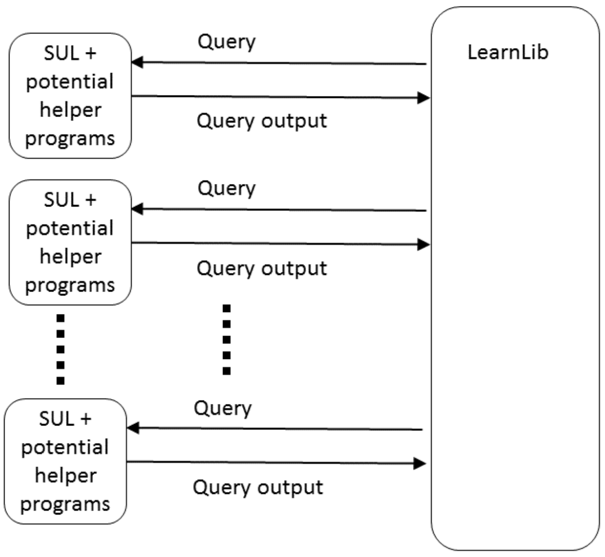

SUL + potential helper programs
Query
Query output
SUL + potential helper programs
Query
Query output
SUL + potential helper programs
Query
Query output
LearnLib

Figure 4: Concept of parallelized learning.

physical effects of a slippery road, the subsequent effects of the software to keep the car on the road will also be tested.
Helper programs could also be tools used for abstraction, such as the Tomte tool [20].

LearnLib now has to make sure that the queries are divided amongst the multiple instances of the SUL, so that they can be executed at the same time as far as possible.

Note that this approach requires multiple CPUs or CPU cores, to run the multiple instances of the SUL simultaneously on.

3.1 Membership queries

As we can see in the description of the L^* algorithm 1, the execution of membership queries takes place during the *update* function. The function is called at three places in the algorithm. The first place can be executed in two cases, in the case of initialisation in which S only consist of ϵ and therefore, the number of cells that has to be filled in the table is the same as the number of membership queries that has to be executed and that is the same as the size of the input alphabet. In the other case this first occurring call to the *update* function is executed after a counterexample has

been found. In that case all prefixes of the counterexample are added to S and probably also elements to $S \cdot I$ are added. Each of these additions adds a new row to the observation table. In most cases this would mean that a lot of membership queries have to be executed at once.
Now let's look at the two other function calls to the *update* function. The second function call is when the algorithm tries to close the observation table. In that case only a single item is added to S, nevertheless this introduces a single new row in the observation table and possibly new rows in the $S \cdot I$ section.
The third call to the *update* function is when the algorithm tries to make the table consistent. In that case a element is added to E and thus a new column is added to the table. This also probably would lead to the execution of multiple membership queries at once.

Note that it doesn't necessarily mean that each new cell in the observation table leads to a new membership query that has to be executed on the SUL, since multiple cells in the observation table can represent the same query and thus the result may already be present in the table. Nevertheless, if a large amount of cells are added at once it is likely that still many membership queries have to be executed at that time.
Since the execution of membership queries during a single call of the *update* function are independent from each other (the execution of such a membership query does not require any outcomes of the other membership queries) they can be parallelized easily. The set of membership queries could be partitioned into several subsets and each subset could be assigned to a different instance of the SUL. We should wait for the completion of the membership queries in each of the subsets, before returning the results to the L^* algorithm, since the results of all queries in a single call to the *update* function have to be returned at once.
If for example 20 membership queries have to be executed and suppose we have 4 instances of the SUL running at the same time, we could execute membership queries 1-5 on SUL #1, membership queries 6-10 on SUL #2, membership queries 11-15 on SUL #3 and membership queries 16-20 on SUL #4.

Note that parallelization of the membership oracle has already been implemented in the new open source version of LearnLib, however it seems to be nowhere mentioned nor evaluated how effective it is in literature.

3.2 Equivalence queries

Instead of executing the random queries, as generated by the random words algorithm, one by one, we also could execute those random queries simultaneously, since they are independent of each other. This is done by generating in advance the maximum number of random queries. That set is divided among all instances and then executed by those instances. Once an instance finds a counterexample, all other instances will be terminated and the found counterexample will be returned to the L^* algorithm.

This method works perfectly to divide the work over multiple instances, however, since the scheduling of the instances is not always exactly the same, this method introduces non-determinism. If the algorithm is executed multiple times, it might be the case that one instance is already further ahead with its tasks compared to another instance, and therefore already discovers another counterexample. That other counterexample will be returned to L^* and subsequently therefore the selection and execution order of membership queries might be influenced.

This does not diminish the correctness of this algorithm. If there exists a counterexample in the set of pre-generated equivalence queries there will be found a counterexample, if no counterexample exists in that set, no counterexample will be found.

Parallelization of the equivalence oracle was not implemented in LearnLib, but could be added in a similar way as the parallelization of membership queries has been implemented.

3.3 Theoretical benefit

According to Amdahl's law [35] the speedup of parallelizing software is defined as follows:
$Speedup = (s+p)/(s+\frac{p}{N}) = \frac{1}{s+\frac{p}{N}}$
where N is the number of processors, s the fraction of time spent on the serial parts of the program and p the fraction of time spent (by a serial processor) on the parts of the program that can be done in parallel. Since $s + p = 1$ this formula can be rewritten as: $Speedup = \frac{1}{(p-1)+\frac{p}{N}}$

It follows that the minimum speedup is 1 in the case that nothing can be done in parallel ($p = 0$). The maximum speedup is equal to N in case that all queries can be done in parallel.

Which fraction can be parallelized (the values of s and p) during the membership queries is dependent on the output that the actual SUL gives and therefore not known in general.

The expected speedup of parallelizing the equivalence queries is N since they can be executed all in parallel. However, this ignores the fact that an individual query can not be parallelized and has has to be fully executed before the equivalence oracle is finished. Thus the executing time of the equivalence queries is at least the time needed for a single equivalence query.

However, in the preceding, the non-deterministic effects of parallelizing equivalence queries are ignored. Dependent of the returned counterexample by the equivalence oracle, the subsequent learning process is influenced. Therefore, which and how many queries will follow might vary. If we would take those effects in consideration, speedups of below 1 (slow downs) are possible in case that much more queries have to be executed than in the baseline situation due to the non-deterministic effects. Speedups of above N are possible in case that much less queries have to be executed than in the baseline situation.

4 Saving and restoring states

The concept of saving and restoring states is based on the idea that reusing previously executed states could contribute to a speed improvement of the learning process. It turned out that our idea of reusing previously reached states of the SUL is not completely new. There has been a study by Bauer et al[29] in which this idea has already been described. However this study makes use of the properties of a specific SUL and is not a good generic solution. A web application that automatically keeps the current state stored in a database was used as a SUL. After finishing the execution of a query they simply do not delete the previous state in the database. During the execution of a subsequent query there is checked whether an existing state in the database could be reused or a new entry has to be made in the database.

This approach has three major drawbacks. First, a state could only be reused once, because after reusing the state of an instance of the SUL its state will change. This still requires a lot of, potentially time consuming, reset actions of the SUL. The second drawback is that if a normal desktop application is used as a SUL, instead of the used database driven web application, all instances of the SUL should remain open, which will cause an enormous usage of RAM. Third, states are not transferable to another computer.

In our approach states of the SUL are stored explicitly to disk. These checkpoints can be restored as many times as necessary. There is no need any more to reset the SUL to start executing a new query. Instances do not have to remain open, so the enormous usage of RAM is neither necessary any more. Checkpoints can be transferred to another computer, which is useful when combining parallelism using multiple computers with saving and restoring states.

4.1 Algorithm

A *trie* (i.e. a prefix tree; a tree where the position in the tree determines the key to the corresponding node) is used to store information about previously executed queries. In figure 5 we could see a small sample of such a trie. Each node consists of an input, a corresponding output and optionally a reference to the stored state of the machine at that point. To search the outcome of a query in the trie, you start at the root node and for every input step in the query you select the corresponding child node. During this process you could find the corresponding outputs for each step of the query. For example the query `tea button > start button` will lead to the outputs `quiescence > tea` in figure 5 and the state of the SUL after executing that query is saved in in checkpoint #2.

During initialization the SUL is started and subsequently an checkpoint of the initial state of the SUL is made. Algorithm 2 describes the process of executing a query using checkpoints.

4.2 Theoretical benefit

As we can see, in the current implementation of the L* algorithm in LearnLib frequently the same input sequences need to be processed by the SUL.

A query consists of a prefix of which the outcomes are already known, because it has been executed before, and a suffix of which the outcomes are not known yet because it has not been executed before.

Algorithm 2 Executing a query

```
if the query is completely present in the result trie (the query has been
executed before) then
    return the corresponding outputs as found in the trie
else
    Find in the result trie the checkpoint $c$ for which the longest part of
    the query has already been executed
    if No such checkpoint can be found in the result trie then
        Use the initial checkpoint
    end if
    Restart the checkpoint $c$ to restore the SUL to this state
    Execute the remaining steps of the query
    Add the observed outputs to the result trie
    if query needs to be saved (see section 4.3) then
        Make a checkpoint of the new state of the SUL
        Add a reference to the new checkpoint in the result trie
    end if
    return the outputs found in the trie that leads to checkpoint $c$
            + the new observed outputs
end if
```

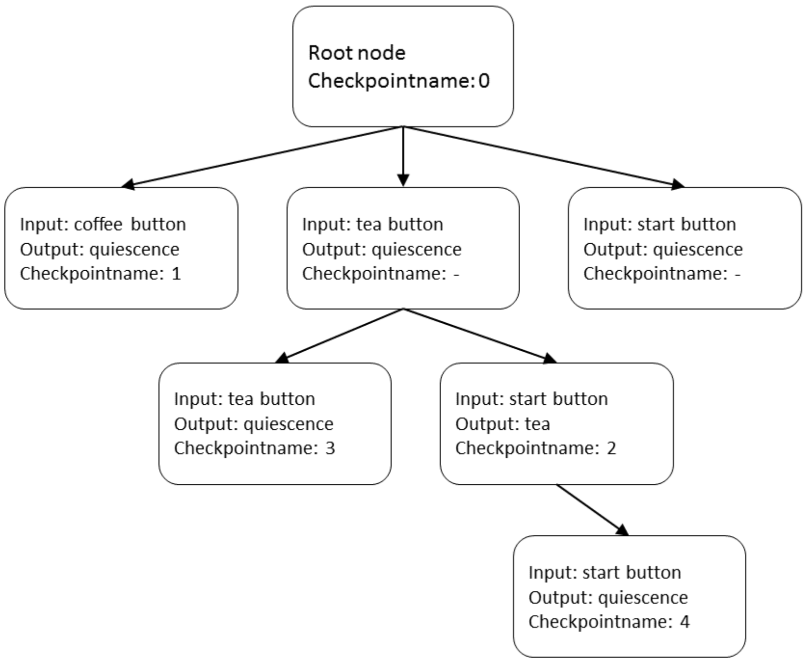


Figure 5: Example of a trie to store the results of previously executed queries.

In the original situation, to execute this query we would have to perform:

1. Reset the SUL - average time costs: $\mathbf{t}_r$
2. Execute the prefix **pf** (here the output of the SUL is completely ignored, it is only executed to bring the SUL in the desired state) - average time costs: $\mathbf{t}_{pf}$
3. Execute the suffix **sf** - average time costs: $\mathbf{t}_{sf}$

When using saving and restoring states of the SUL instead you could perform:

1. Restore the saved state of the SUL after prefix **pf** was executed- average time costs: $\mathbf{t}_l$
2. Execute the suffix **sf** - average time costs: $\mathbf{t}_{sf}$
3. Save the new state of the SUL - average time costs: $\mathbf{t}_s$

Thus to have benefit from saving and restoring states it should be true that:

$$\mathbf{t}_l + \mathbf{t}_{sf} + \mathbf{t}_s < \mathbf{t}_r + \mathbf{t}_{pf} + \mathbf{t}_{sf}$$

$$\mathbf{t}_l + \mathbf{t}_s < \mathbf{t}_r + \mathbf{t}_{pf}$$

It seems likely that an increase of the size of the model causes an increase of the average query length. Consequently, the average length of the queries would be bigger and therefore the average time to execute the prefix: $\mathbf{t}_{pf}$ will also increase, while $\mathbf{t}_l$, $\mathbf{t}_s$, $\mathbf{t}_r$ will probably remain the same. This leads to the following hypothesis: When the size of the SUL increases it will become more beneficial to apply techniques to save and restore states of the SUL.

Note that in the formula above, we assume that the state of the SUL after each query is saved. This is not optimal, since equivalence queries likely consist of long series of randomly chosen inputs. Therefore the chance that the state after executing an equivalence query can be reused will be very small. Since in real life situations it can happen that even more equivalence queries are needed than membership queries to learn a correct model[42], and thus a lot of time would be wasted on saving states that probably won't be reused, we decided to only apply the saving of states after executing membership queries. However, during the execution of all these equivalence queries there still can be taken advantage of the saved states of the membership queries.

4.3 Selection of states which should be saved

Although it is decided to only save the states after executing membership queries, there is still a lot of time and memory spent on saving states. A lot of time might be saved by reducing the number of states to be saved. Instead of:

1. Restore the saved state of the SUL after prefix **pf** was executed - average time costs: $\mathbf{t}_l$
2. Execute the suffix **sf** - average time costs: $\mathbf{t}_{sf}$
3. Save the new state of the SUL - average time costs: $\mathbf{t}_{save}$

The following should be applied:

1. Restore the saved state of the SUL where the largest part of prefix **pf** already has been executed - average time costs: $\mathbf{t}_l$
2. Execute the remaining part of prefix **pf** (here the output of the SUL is completely ignored, it is only executed to bring the SUL in the desired state) - average time costs: $\mathbf{t}_{rpf}$
3. Execute the suffix **sf** - average time costs: $\mathbf{t}_{sf}$
4. In some cases: save the new state of the SUL - average time costs: $\mathbf{t}_{save}$

This will reduce the total time costs of saving states, but introduce more time costs of executing the remaining parts of the prefixes **pf**. This will lead to the following hypothesis: there is a trade-off between the costs of making more checkpoints and the benefit that those extra checkpoints offer. Therefore there will be an optimum, which checkpoints need to be saved to get the fastest execution time. Neither saving none nor saving all possible states of the SUL is likely to be optimal in most cases. Therefore we would like to know where the optimum is.

On the one hand will be that the longer the query is the smaller the expected reuse of the saved state of that query will be. (For example a checkpoint of a query with a size of 1 could be used by much more queries than a checkpoint of a query with a size of 10). Therefore you would expect a strategy that saves more smaller queries will be more useful. On the other hand saving larger queries might be more useful because if you reuse such a saved state the benefit at a time will be bigger. It is very difficult to theoretically determine precisely the optimal strategy in general without having information about how many queries of which length are executed. This depends on the actual SUL being used. So we decided to empirical evaluate the effect of a couple of different strategies, see section 6 and section 7. The following strategies will be researched;

1. Using a linear function. Instead of saving every membership query we could for example save only membership queries with a size that is a multiple of 2. A naïve estimation (ignoring the fact that the query size is not likely to be equally distributed) is that the total time to save queries will probably be about halved $\frac{1}{2} \cdot \#mq \cdot \mathbf{t}_{save}$ instead of $\#mq \cdot \mathbf{t}_{save}$. On average there probably will be an additional $\frac{1}{2}$ step needed to execute a query. In total $\frac{1}{2}(\#mq + \#eq)\mathbf{t}_{step}$ additional time is needed to execute all the queries, where $\mathbf{t}_{step}$ is the average time needed to execute a single step.
   If we would save only the queries with a size that is a multiple of 3 the benefit will be probably about: $\frac{2}{3} \cdot \#mq \cdot \mathbf{t}_{save}$ while there is a disadvantage of about $(\frac{1}{3} + \frac{2}{3})(\#mq + \#eq)\mathbf{t}_{step}$
   In general we would expect a benefit of $\frac{n-1}{n} \cdot \#mq \cdot \mathbf{t}_{save}$ and a disadvantage of $(\sum_{i=1}^{n-1} 1/i)(\#mq + \#eq) \cdot \mathbf{t}_{step} = \frac{n-1}{2}(\#mq + \#eq) \cdot \mathbf{t}_{step}$

2. Setting a maximum value for the size of a membership query to be saved.
   The expectation is that mainly the execution of equivalence queries will benefit from this reduction. There are i^l possible queries, where i is the number of inputs and l is the length of the query. Suppose there are 10 inputs, then there are only $10^3 = 1000$ possible queries with length 3. It is feasible that a large part, or even all of these 1000 possible queries are executed during the execution of membership queries and the resulting states of the SUL are saved. Therefore, if during the execution of equivalence queries a query is randomly chosen, the chance that it can reuse such a saved state will be large. But if you would look at queries with length 9 then there are 10^9 possible queries. Since it is not likely that a billion states are saved, the chance that a randomly chosen equivalence query shares the same prefix of 9 inputs with those of a saved state will probably be very small. This will lead to the hypothesis that randomly chosen equivalence queries will only profit from saving states of small queries.

3. Setting a minimum value for the size of a membership query to be saved.
   This strategy is opposite to the aforementioned method, but nevertheless it might be beneficial. Because if a large query could be reused, the individual benefit of reusing the accompanying saved state will be larger, since the execution of more steps is superseded. The hypothesis is that this benefit will only appear during the execution of membership queries, since membership queries are built upon previously executed membership queries.

4. Using an exponential function saving only queries with a size of 1, 2,4,8,16. . . steps.
   The hypothesis is that this method will combine the benefits from having saved the state of small queries which is useful for randomly chosen equivalence queries and having saved the state of large queries in which the individual profit of reusing especially during membership queries will be large. Nevertheless still a lot less saved states are needed than when simply every executed query is saved. Now only $2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \ldots + 2^x = 2^{x+1} - 1$ different query lengths are saved. This is a lot less than storing all the $1{+}2{+}3{+}4{+}\ldots{+}2^x = \frac{2^x+1}{2}2^x$ different query lengths.

5 Implementing saving and restoring software states in practice

The idea of saving and restoring states of an arbitrary SUL will lead inevitably to the question how these states can be saved and restored, since this is not trivial functionality that is by default available in a lot of SULs. Luckily, there do exists some possible solutions for this problem. In this thesis we will focus on application checkpointing using DMTCP.

DMTCP (Distributed MultiThreaded CheckPointing) [27] is a Linux framework that allows to save and restore the state of a group of arbitrary Linux processes. DMTCP makes basically a kind of RAM dump of a process. DMTCP runs fully in user-space and does not require any modifications of the Linux kernel neither of the applications of which the state is stored. DMTCP was originally designed for cluster computing. In case that a large workload, for example a large scientific computation that takes 30 days to

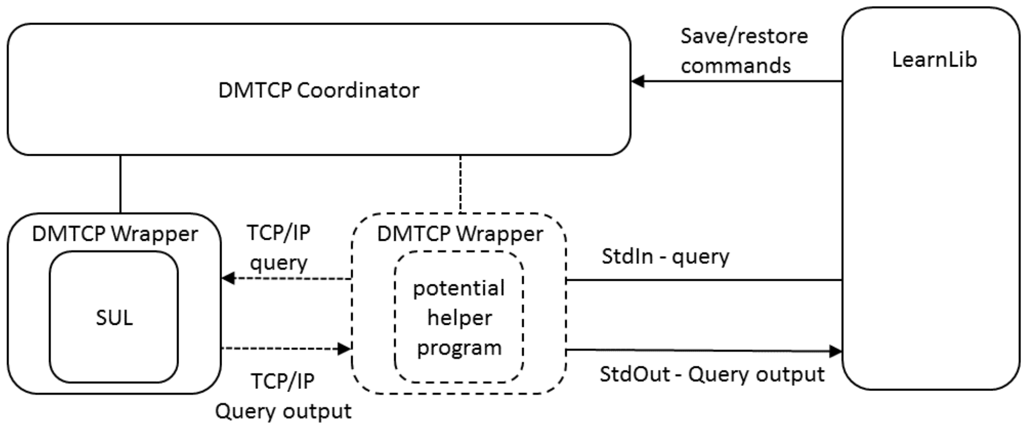


Figure 6: Combining LearnLib with DMTCP

execute crashes after 29 days due to a hardware failure, it can be ensured that those 29 days of computational work are not lost. Instead of restarting the whole computation from the very beginning, a checkpoint which was made at for example day 28 could be restarted, such that the whole computation could be finished in only 2 additional days. Since it is focussed on cluster computing, checkpoints are transferable, thus if they are made at one computer, then they can be restarted at another computer. Apart from cluster computing DMTCP has already successfully been deployed for, for example, a universal reversible debugger [48]. It has also been applied recently in the field of model checking of distributed systems to handle the problem of non-deterministic output. [38]

Besides the fact that it has been applied successfully in other studies (although for a different purpose) we chose DMTCP because it is easy to implement, it is reasonably fast and although it is still experimental it is quite stable and it offers a quite generic solution. Many Linux applications can run under DMTCP. Although automata learning is especially useful in the field of embedded software, it doesn't mean that the software always requires a real-time OS. Less time-critical software may run natively on Linux and otherwise the software may run in a simulated environment that runs on Linux.

Figure 6 shows how DMTCP is integrated in learning a SUL with LearnLib. To run DMTCP the coordinator application has to be started. This coordinator is able to checkpoint all programs that are running under it, and to restore a previously saved checkpoint. Subsequently, the application to be checkpointed has to be started with a DMTCP wrapper. Multiple applications can run under the same coordinator. Through the operation of DMTCP, TCP/IP sockets, pipes, file descriptors, mutexes/semaphores all can be used without problems. However, at the moment a checkpoint is made, there should not be an open TCP/IP connections between applications that run under the coordinator and the outside world, but an open TCP/IP connection between two programs that run under the same coordinator will not be a problem. A standard input/output connection will never give any problem with checkpointing. Suppose there is a SUL that communicates using a socket connection, then DMTCP could be applied by writing a simple helper application, which redirects the socket connection over standard input/output, runs under the same DMTCP coordinator, and thus is also stored in each checkpoint. This situation is shown in figure 6. If the SUL can communicate over standard input/output itself, this helper program is not needed of course.

The run-time overhead of DMTCP is negligible. DMTCP only applies wrappers around some less frequently used system calls, such as *open()*, *getpid()* and *socketpair()*. More frequently used system calls such as *read()* and *write()* are not interfered by a wrapper [4].

Note that it is possible to run several DMTCP coordinators at the same time on the same computer. We use this to combine DMTCP with parallelism.

6 Evaluation method

6.1 Experiments

The proposed speed improvement techniques, parallelism and checkpointing using DMTCP, have been implemented in practice and integrated using Java in the new open-source version of LearnLib[11]. Here we used an own data structure to implement the result trie (See section 4.1) to keep track of all previously executed queries and stored checkpoints. This data structure has been implemented separate from the existing data structures in LearnLib. In order to evaluate both speed improvement techniques we have done some experiments using three sample systems: the Bounded Retransmission Protocol, Tic-Tac-Toe and a simulation of the Engine State Machine.

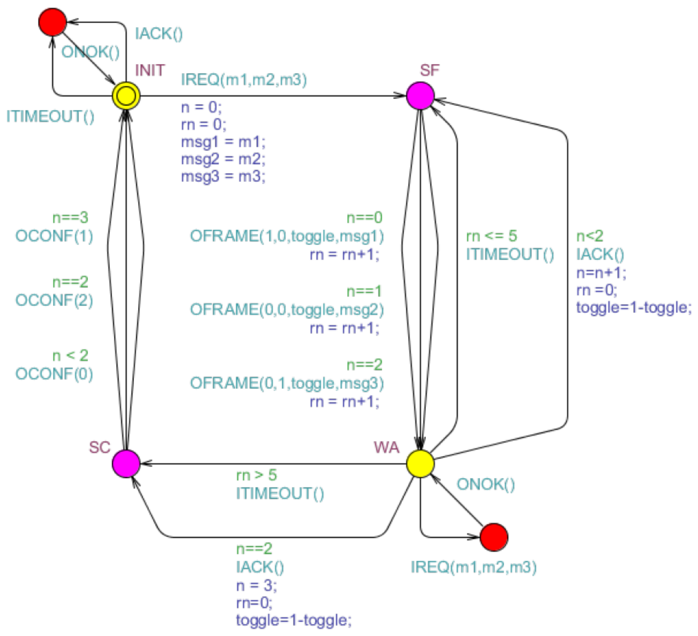
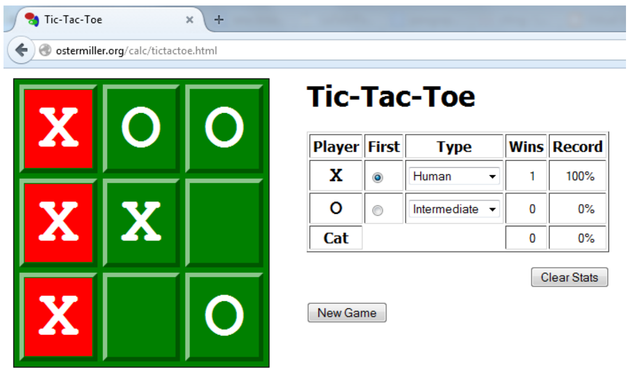


Figure 7: State diagram of the Bounded Retransmission Protocol. Figure from [24].

6.1.1 Bounded Retransmission Protocol

The Bounded Retransmission Protocol is a well-studied protocol originating from Philips for communication between an infrared remote control and a television set. Figure 7 shows the state diagram of the protocol. This protocol has already been used before to learn models with LearnLib. Hereby the side of the sender was learned [24]. This SUL is simulated in a very simple C++ program, which consists of only 1 file with a couple of switch statements. Communication takes place using standard input/output. The tricky part of learning this protocol is to find counterexamples where all the outputs $OCONF(0)$, $OCONF(1)$ and $OCONF(2)$ appear. In order to achieve that, the number of retransmissions must become high enough. In our case this retransmission number is set to 5. The possible values of each message are restricted to 2.

Some statistics about this SUL:



Tic-Tac-Toe

Player	First	Type	Wins	Record
X	●	Human	1	100%
O	○	Intermediate	0	0%
Cat			0	0%

Figure 8: Ostermiller Tic-Tac-Toe webpage

number of inputs	10
number of possible outputs	18
number of states in a correctly learned model	155
default (average) time to process a single step	10ms
default (average) time to reset the SUL	20ms
RAM usage of the SUL	10MB
average size of a DMTCP checkpoint	16MB
average time to take a DMTCP checkpoint	75ms
average time to restore a DMTCP checkpoint	20ms

From figure 7 we can derive that to observe the $OCONF(0)$ output a counter example of at least 7 inputs is needed ($IREQ$ and subsequently 6 times $ITIMEOUT$). To observe the $OCONF(1)$ output a counter example of at least 4 inputs is needed ($IREQ$ and subsequently 3 times $IACK$) and to observe the $OCONF(2)$ output a counter example of at least 10 inputs is needed ($IREQ$, subsequently 2 times $IACK$ and finally 7 times $ITIMEOUT$). As equivalence oracle the parallelized version of random words is used with a minimum query length of 10 and a maximum query length of 15, with maximum 10000 queries per round. These settings turned out, during all test runs, to be good to find the needed counter examples and learn a correct model of the SUL.

6.1.2 Tic-Tac-Toe

The Ostermiller Tic-Tac-Toe webpage[8] contains a JavaScript implementation of the well known Tic-Tac-Toe game. We use the setting that lets a human play against the computer.

As inputs we use all the 9 different cells in which the human user could place an X. As outputs we use all the 9 different cells where the computer subsequently places the O plus the quiescent output (in case that the input corresponds to a non-empty cell or in case that the game has ended)

If only empty cells are selected as input, no more than 5 inputs are possible in sequence to end the game and in the case a non-empty cell is selected as input the state of the SUL will not change, which makes this a manageable SUL.

In the original source code a random function is used to the make the game less predictable and more fun to play. But this random function makes the game non-deterministic and therefore unable to be used with the L* Algorithm. An adjustment is made to replace the random function with a function that produces the same semi-random output after each new game, which makes this SUL deterministic.

To communicate between LearnLib and the SUL the web test automation software Selenium[13] is used, which is able to send click events to the web-page in order to send an input to the SUL and retrieve HTML attributes from the webpage to forward outputs to LearnLib. The HTMLUnit web-driver (GUI-less browser in Java) is used for rendering the webpage. This makes it usable with DMTCP to save and restore checkpoints of the SUL without having to perform tricks to checkpoint GUI applications.

Communication takes again place using standard input/output. To run this Java based SUL the compact Java virtual Machine JamVM is used [7]. This reduces the RAM usage by around 10% to 20% compared to the default OpenJDK virtual machine, which makes checkpoints smaller and faster to save and restore.

Some statistics about this SUL:

number of inputs	9
number of possible outputs	10
number of states in a correctly learned model	148
default (average) time to process a single step	163ms
default (average) time to reset the SUL	226ms
RAM usage of the SUL	78MB
average size of a DMTCP checkpoint	138MB
average time to take a DMTCP checkpoint	355ms
average time to restore a DMTCP checkpoint	101ms

A game will end after at most 5 inputs. However if a non-empty cell is selected as input, the required number of inputs to end the game will increase. Thus counter examples with a length of above 5 are needed to learn a correct model.

As equivalence oracle the parallelised version of random words is used with a minimum query length of 7 and a maximum query length of 15, with maximum 5000 queries per round. These settings turned out, during all test runs, to be good to find the needed counter examples and learn a correct model of the SUL.

6.1.3 Simulation of the Engine State Machine

The Engine State Machine (ESM) is a piece of embedded software developed at Océ. It controls transitions between states in a printer. This is a nice example of real embedded software. There has already been extensively researched how a model of this software could be learned using LearnLib [42]. Instead of using the actual ESM a simulation is used. This Java based simulator reads a text file with the description of a state machine (a Graphviz DOT file) and acts like it is the actual machine from the description. Communication takes again place using standard input/output.

Learning the whole SUL with over 4000 states took more than 287 hours in[42] and in our case we want to test several strategies which require several test runs. In addition, our simulation of the ESM reacts slower than the actual ESM, as used in [42], which would make learning the complete model with over 4000 states infeasible within a reasonable amount of time. Therefore a very small subset is taken, by using only a couple of inputs. This drastically reduces the size of the model to be learnt. Unfortunately, this makes directly comparing our evaluation results with those found in [42] impossible.

Some statistics about this SUL:

number of inputs	7
number of possible outputs	149
number of states in a correctly learned model	23
default (average) time to process a single step	50ms
default (average) time to reset the SUL	500ms
RAM usage of the SUL	83MB
average size of a DMTCP checkpoint	81MB
average time to take a DMTCP checkpoint	262ms
average time to restore a DMTCP checkpoint	105ms

Note that the size of a checkpoint is slightly smaller than the RAM usage. This can be explained by the functionality of DMTCP to reduce the size of a checkpoint if a memory region contains only zeroes.

As equivalence oracle the parallelised version of random words is used with a minimum query length of 5 and a maximum query length of 15, with maximum 2000 queries per round. These settings turned out, during all test runs, to be good to find the needed counter examples and learn a correct model of the SUL.

6.2 Learning set-up

For the execution of the experiments we used a (super)computer with in total 60 CPU cores running at 2.3GHz, 3TB RAM and the availability of RAID storage. This machine is ideal to evaluate our proposed optimization techniques, since the large number of CPU cores makes it possible to run many instances of a SUL at the same time which gives good insight in the scalability of our parallelization technique. This machine also features fast file storage, which reduces the time to take and restore checkpoints. For optimal storage speed, we used a RAM disk, with the ability to use the RAID storage in case the RAM disk is full.

6.3 Test run selection

1. First we evaluate the effect of parallelization on all 3 test systems.
2. Then we evaluate the effect of using DMTCP without parallelization. For this we will apply the 4 proposed strategies to reduce the number of checkpoints.
3. Thirdly, we will measure the benefit of combining parallelization with saving and restoring states. To reduce the number of measurements, only the best performing strategy is chosen in combination with 60 parallel instances (the highest number of parallel instances that the computers allows to be executed fully in parallel)
4. In section 4.2 we have seen that in theory the benefit of saving and restoring states would be bigger when the SUL responds slower. Since we are also interested in how beneficial the proposed optimizations are

on slower responding systems we will vary the response time of the previous mentioned SULs by slowing them down (using a sleep function). To reduce the number of measurements (which might have required several weeks of execution time in total) these slowed down versions of the SULs are only executed using the best performing strategy to reduce the number of checkpoints. This does not automatically mean that this strategy is also the best performing strategy on the slowed down versions, however, evaluating all strategies again on the slowed down SULs would take too much time for this thesis, and we think that the best performing strategy on the non-slowed down SULs would still perform reasonable on the slowed down SULs.

For each of the slowed down SULs we measure a new baseline, the execution time with only DMTCP applied, the execution time with 60 parallel instances and finally, the execution time with both DMTCP and 60 parallel instances.

7 Results

In this section only the summary of the results are shown. The complete overview of results, including all details, can be found in Appendix A.

7.1 Parallelization

As we can see in Figure 9, applying parallelism is very effective to improve the learning speed. In most cases, doubling the number of parallel instances nearly doubles the execution speed; on average the speed will increase with a factor 1.83.

Note that there is a variation in the number of membership and equivalence queries. This is caused by the way our parallel equivalence oracle algorithm works. Due to the non-deterministic effects of process scheduling of the parallel instances different test runs can result in different counter examples. Therefore the subsequent learning process can be influenced which can result in more or less queries that has be executed. This explains the outlier of the ESM with 16 parallel instances.

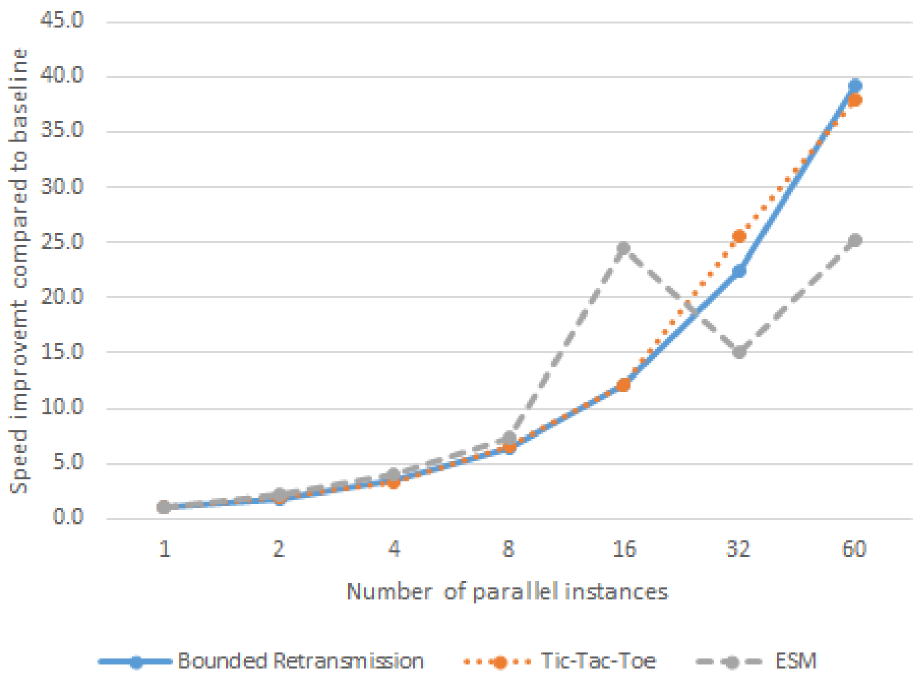


Figure 9: Results of parallelization

7.2 DMTCP

If only DMTCP is applied without any parallelism, the execution is fully deterministic. Each test run will have exactly the same (number of) queries and will always produce exactly the same output. Thus in the following results it is not needed to take any effects of non-determinism, such as varying query counts, in account.

7.2.1 Strategy 1: limiting checkpoints using a linear function

In figure 10 we can see that it is indeed not optimal to store every membership query nor saving no query at all. This strategy is effective, at the optimum (which lies around a multiplication factor of 2 or 3, i.e. where only queries with a size of 2,4,6,8,. . . or with a size of 3,6,9,12,. . . are stored) there is a significant speed improvement over using no DMTCP at all. However, choosing the wrong multiplication factor might slow down the learning process compared to the baseline, because in that case the benefit of having to execute less inputs on the SUL does not compensate for the additional time the saving and restoring of checkpoints takes. The dip in the graphs

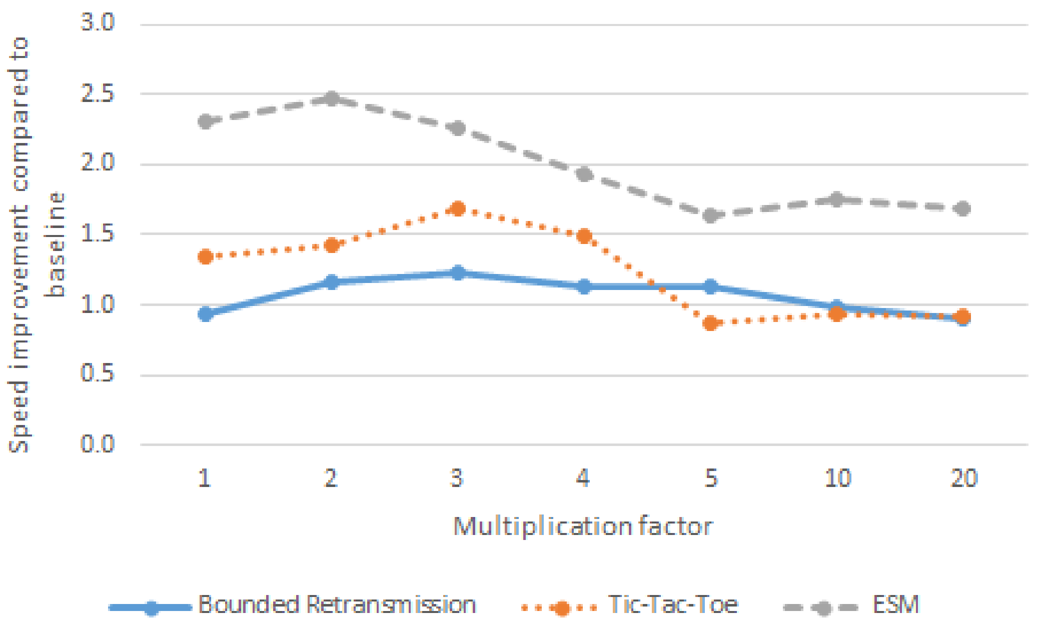


Figure 10: Results of strategy 1

of Tic-Tac-Toe and ESM at a multiplication of 5 could be explained by the distribution of queries based on their size. If for example much more queries with a size of 4 and 8 could be reused than with a size of 5 and 10 it is logical that the speed improvement of a multiplication factor of 5 is lower than a multiplication factor of 4. The slightly speed increase with a higher multiplication factor than 5 could be explained by that a lot less checkpoints have to be made and the average benefit of reusing a single checkpoint increases.

7.2.2 Strategy 2: limiting checkpoints using an upper limit

Also this strategy can be effective as we can see in figure 11. It depends on the SUL where the optimum is. For the Bounded Retransmission Protocol it is less effective than the previous strategy, for Tic-Tac-Toe it is more effective than the previous strategy. A possible explanation is that the checkpoints of the Bounded Retransmission Protocol are much smaller, which makes the overhead of also saving larger queries with a size of 6,9,12,15,... smaller and therefore more beneficial even if those larger queries could be reused only once. The overhead of saving queries of the Tic-Tac-Toe SUL is much larger (because it uses much more RAM), so it is more beneficial to save only the

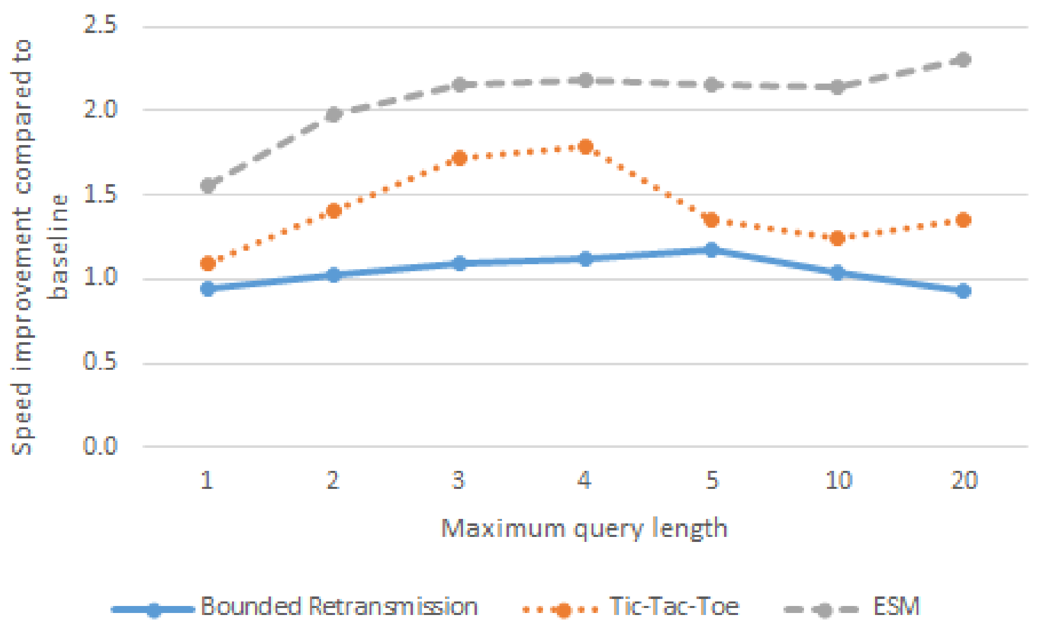


Figure 11: Results of strategy 2

first couple of small queries, which could be reused much more often. For the ESM it turns out to be faster to have the highest maximum size as possible, thus it is better to save all queries than using an upper limit (The longest query size used to learn the ESM model is 20). However it can still be an effective way to reduce the number of checkpoints.

7.2.3 Strategy 3: limiting checkpoints using a lower limit

For this strategy, we can reject our hypothesis, this strategy only worsened the execution speed regardless of the settings as we can see in figure 12. In addition it is also not an effective way to reduce the number of checkpoints while maintaining a relative good execution speed.

7.2.4 Strategy 4: limiting checkpoints using an exponential function

The speed compared to the baseline of this strategy is respectively: 1.19 for the bounded retransmission protocol, 1.70 for Tic-Tac-Toe and 2.14 for the

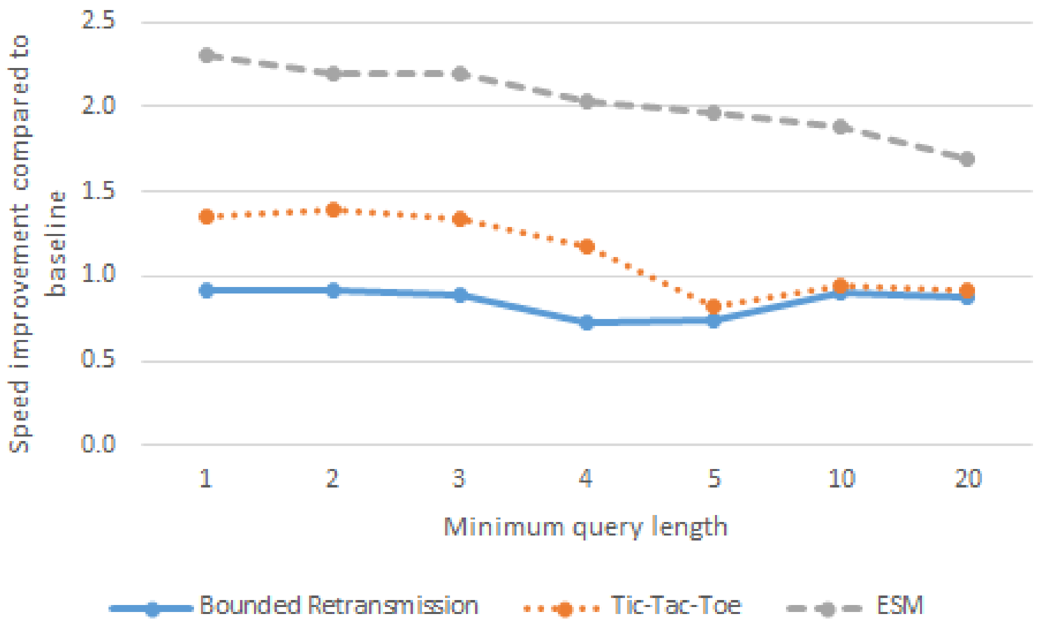


Figure 12: Results of strategy 3

ESM model. So, this strategy does work to improve the execution speed.

7.2.5 The best performing strategy

In figure 13 we could see the performance of all 4 strategies, here the average is taken of all 3 test systems (Bounded Retransmission Protocol, Tic-Tac-Toe and ESM). In this chart strategy 3 seems to perform slightly worse than strategy 1 and 2, but we have seen that the only best settings are reached when all queries are saved, which makes this strategy not useful. The performance of strategy 4 is around the best settings of strategy 1 and 2. Only for the ESM the speed is a bit behind of the optimal setting of strategy 1, but the number of checkpoints is lower. Since strategy 4 does not require any fine-tuning of settings and still performs good we prefer this strategy.

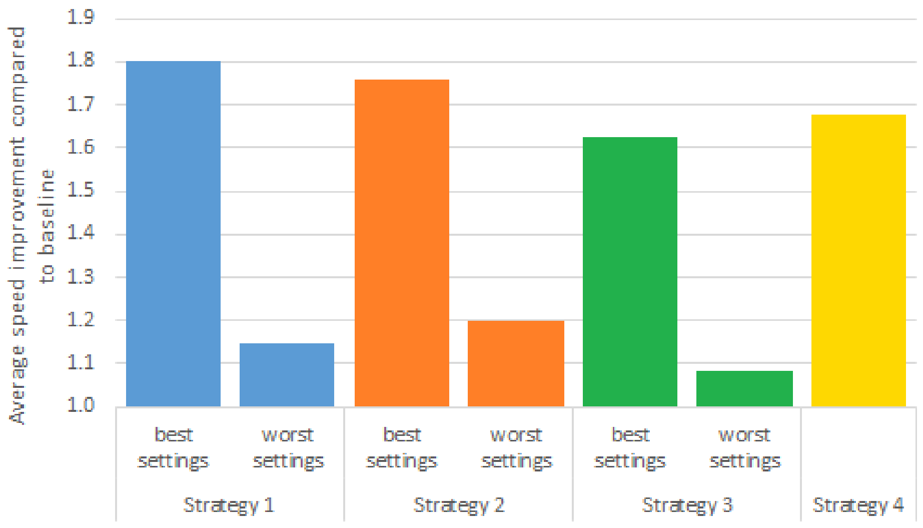


Figure 13: Comparison of the performance of all 4 strategies, the average of all 3 test systems is taken

7.3 Combining parallelism with checkpointing

Here we combine 60 parallel SULs with checkpointing using the exponential function (strategy 4).

Tic-Tac-Toe and the ESM show a significant improvement over only applying parallelism or only applying checkpointing. Tic-Tac-Toe becomes 40.80 times faster than the baseline and the ESM becomes 42.17 times faster than the baseline. However, the bounded retransmission protocol shows a significant worsening compared to only applying parallelism with 60 SULs running in parallel. There is a speed improvement of 14.79 (parallelism plus checkpointing)versus 44.04 (only parallelism). A possible explanation is that when a lot of checkpoints must be saved and restored at the same time, which is the case due to the very low step and restart times, i.e. the SUL is very fast, the available disk speed must be shared across all these parallel instances that want to read and write to the disk at the same time. Therefore the disk speed will become a bigger bottleneck. This is conformed when the SUL is slowed down, as we can see in section 7.4.

7.4 Slowing down the SULs

Bounded Retransmission:
The bounded Retransmission Protocol is slowed down so that a reset takes 2000ms (instead of 20ms) and a step takes 20ms (instead of 10ms).
Note that the mentioned baseline is the baseline of this slowed down version, which is 18.03 times slower than the original baseline.

	baseline	DMTCP	parallelism	both combined
Speed compared to baseline	1.00	15.44	47.91	297.62

Tic-Tac-Toe:
The Tic-Tac-Toe game is slowed down so that a reset takes 5000ms (instead of 226ms) and a step takes 500ms (instead of 163ms).
Note that the mentioned baseline is the baseline of this slowed down version, which is 8.48 times slower than the original baseline.

	baseline	DMTCP	parallelism	both combined
Speed compared to baseline	1.00	4.70	34.89	97.68

Engine State Machine:
The Engine State Machine simulation is slowed down so that a reset takes 2000ms (instead of 500ms) and a step takes 100ms (instead of 50ms).
Note that the mentioned baseline is the baseline of this slowed down version, which is 3.02 times slower than the original baseline.

	baseline	DMTCP	parallelism	both combined
Speed compared to baseline	1.00	3.95	25.68	78.62

As we can see, combining parallelism with DMTCP is very effective. Depending on the SUL and how slow the SUL reacts it can lead to impressive speed improvements of up to about 300 times faster than the baseline.

8 Evaluation

Based on the results from the previous section we can derive a number of generic recommendations:

- Parallelize as much as possible. Except for one test run probably caused by the non-deterministic effect of parallelism, all our test runs have shown a huge speed improvement when using parallelism. With an average speed improvement of 1.83 when doubling the number of instances, this approach is definitely worth applying. Although in practice it requires many available CPU cores, but these may be spread across multiple computers.

In addition, it must be possible to run multiple parallel instances of the SUL and potential helper programs, which might not be always be the case due to technical as well as license restrictions.

Finally, this optimization ensures that the work is carried out faster, but the amount of work will not be reduced, thus also the total required power consumption of all CPUs to learn the model will not be reduced.

- If the SUL responds slowly, i.e. it takes a long time to reset and a long time to take a step (order of magnitude of hundreds of milliseconds or more for a single step or reset operation), we recommend to apply the use of checkpointing. DMTCP is an effective way to implement checkpointing. It provides a generic solution (for applications that can run under Linux), turned out to be stable although it is still experimental and is reasonable fast in saving and restoring checkpoints. If DMTCP is applied for checkpointing, and a very fast storage device is used, such as a RAM disk, to save the checkpoints and if a SUL reset and step take in the order of magnitude of hundreds of milliseconds, the execution speed might me doubled. However, if it takes several seconds the speed increase will be much larger, possibly even more than 10 times faster than the baseline. On even slower SULs than that we expect it to be even more beneficial.

  To limit the number of checkpoints, it turns out to be the most convenient to make use of the strategy which limits the number of checkpoints using an exponential function. This strategy belongs to the best performing in most cases, without the need for fine-tuning parameters. Only in exceptional cases, for example when there is very little disk space to store the checkpoints, we recommend to consider other strategies.

  Note that using a fast responding SUL and/or a wrong strategy to limit the number of checkpoints, the execution speed can be worsened compared to the baseline.

  Obviously it is required that the SUL can be checkpointed using DMTCP, additionally, it might be needed to write a helper program to prevent problems with open socket connections of the SUL.

- If it is possible, we recommend to combine parallelism with checkpointing, considering that we have seen speed improvements of up to nearly 300 times with this combination. However, if this combination is used with a fast responding SUL it can slow down the learning speed compared to using only parallelism, even if, in the same case, applying checkpointing without parallelism is beneficial compared to the baseline. (See section 7.3)

9 Alternative Techniques

9.1 Alternatives to DMTCP

In this section we will discuss some alternatives to DMTCP.

Ideally we would like to have a solution to save an restore states that is:

- a generic solution, where the SUL is not restricted to a single operating system or restrictions such as file handlers, open network connections, having a GUI or not. DMTCP is a quite generic solution, however it only works for Linux applications, a solution that also works for other operating systems would be better.
- as fast as possible in saving and restoring states; Since we apply this technique to get speed improvement, the overhead should be as low as possible. Although DMTCP turns out to be reasonable fast, there is still room for improvement.
- Checkpoints as tiny as possible. This not only leads to faster saving and restoring of states (in many cases the disk speed is the bottleneck), but it also leads to less required disk space. Therefore the costs of additional hard disks, solid state drives or RAM disks are reduced, or a smaller but faster storage device could be used (solid state drive instead of a hard disk or a RAM disk instead of a solid state drive). In addition it also provides the ability to store more checkpoints on the same storage device, which could be useful when learning large SULs. The checkpoints of DMTCP are quite large in general, in most cases they are much bigger than the RAM usage of the SUL. Alternative solutions might perform much better at this point.

- transferable checkpoints; This means that checkpoints made on one computer can be restarted at another computer. This is useful when combining parallelism using multiple computers with saving and restoring states. DMTCP is developed with distributed computing in mind, therefore it completely fulfils this requirement, but that might not be the case with all alternative solutions.

9.1.1 Virtualization

Software like VMWare[18], Hyper-V[2] and VirtualBox[17] is well-known for hardware virtualization. These programs simulate a complete virtual computer that runs a separate operating system. Consequently, this provides a very generic solution, because it does not restrict SULs to run on a specific operating system. The virtual computer has the same architecture as the host, so no full emulation of processor instructions is necessary. This in combination with x86 hardware virtualization support makes running software in the virtual machine relatively not much slower than running it directly on the host.
These virtualization programs have the ability to suspend the state of a virtual machine and to resume later such a suspended state. These suspended states are files that contain among others the content of the virtual RAM and CPU registers. In itself this already provides a sufficient ability to save and restore the state of an arbitrary SUL that can run inside the virtual machine. In order to do so, manually keeping an administration of states and copying back and forth the suspended state files would be required
In addition, some of these programs also provide support for snapshots. Snapshots are stored states of the virtual machine of which the virtualization program keeps an administration and offers the ability to revert a specific snapshot. This makes implementing it to work with LearnLib easier.
We chose to use VirtualBox in this thesis to perform our experiments with. VirtualBox is open source software, easy to use, has the ability to use snapshots, provides scripting capabilities (using the VBoxManage command-line interface) and it seemed that it was not significantly faster or slower than comparable virtualization solutions.
Figure 14 shows how LearnLib communicates with VirtualBox and the SUL that runs inside the virtual machine. We use the VBoxManage command-line interface to pass commands to VirtualBox. In order to make a snapshot only a single command with the desired snapshot name has to be given. In order to restore a snapshot 3 commands have to be given to the

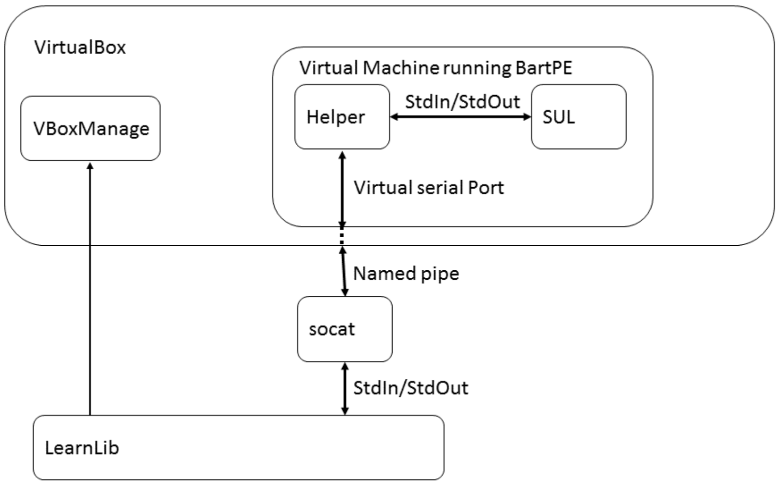


Figure 14: Combining LearnLib with VirtualBox

VBoxManage application: shut down the virtual machine, revert the desired snapshot and resume the virtual machine.

In addition, we need communication between LearnLib and the SUL running inside the virtual machine. In order to accomplish this, we use the virtual serial port of the virtual machine. Instead of for example a virtual Ethernet port, no drivers need to be installed and no re-initialization after resuming a snapshot is necessary. Inside the virtual machine, the required inputs for the SUL and the corresponding outputs of the SUL can be transmitted over this virtual serial port. Since the actual SUL does not communicate over a serial port in most cases a helper program is created that converts the stdin/stdout communication to communication over the serial port. VirtualBox converts this virtual serial connection outside the VM to a named pipe. Using Socat[15] this named pipe is converted to stdin/stdout communication.

Each time when creating a snapshot, the entire state of the RAM, including the operating system and all running applications, is stored to disk. To speed up this process it is convenient to use an as lightweight operating system as possible and to reduce as many unnecessary background applications as possible. Despite the fact that VirtualBox can store and revert the state of a virtual hard disk, doing so may significantly increase the time to take and restore a snapshot. Therefore we chose to do not use a virtual hard disk at all. As operating system we use BartPE[1], which is based on a Windows-NT kernel, can run from a live cd, is able to run many Windows programs and has a memory footprint of only a few dozen megabytes. The SUL and helper program (which automatically starts when after booting) are also stored on a live-cd.

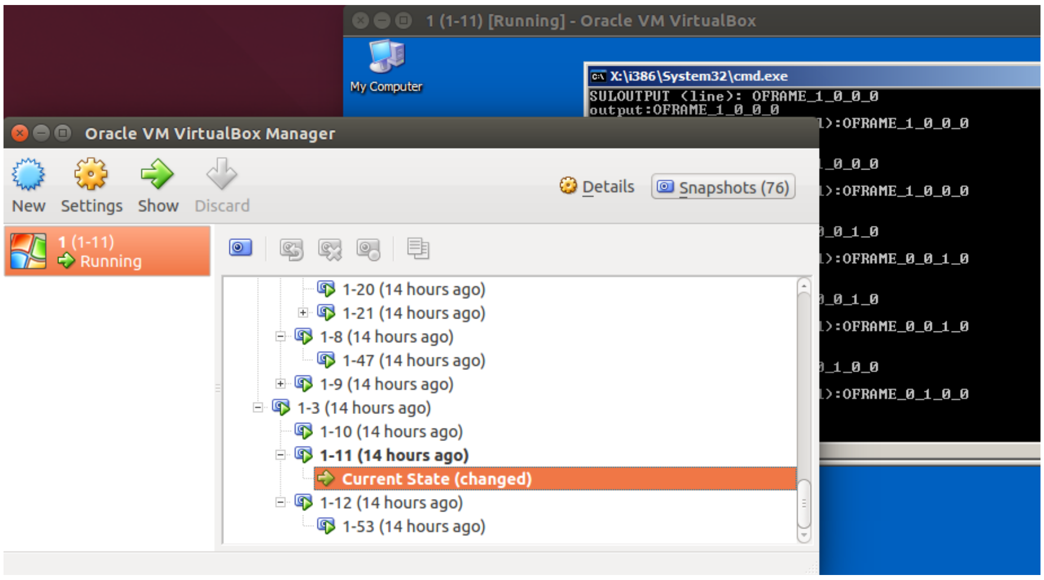
We used VirtualBox to learn a model of the Bounded Retransmission protocol. In contrast to the previous experiments a less powerful computer was used (2.1GHz dualcore, 3GB RAM, 256GB SSD). To learn a correct model within a reasonable amount of time we reduced the retransmission number to three instead of five and used only three inputs instead of ten. The time to take a step is set to 1000ms and the time to perform a reset is set to 2000ms. It turns out that VirtualBox can be used successfully to reduce the execution time, however it is not as efficient as DMTCP in the same situation, as we can see in the following table:

	baseline	with DMTCP	with VirtualBox
#MQ	545	545	545
Time MQ (s)	4020	2217	2760
#EQ	504	504	504
Time EQ (s)	3913	1207	1772
#checkpoints/snapshots	N/A	76	76
checkpoint/snapshot size (MB)	N/A	16	56
Speed compared to baseline	1.00	2.32	1.75

Figure 15 shows the snapshots taken to learn this model. You could see the tree structure, which corresponds to the result trie as described in section 4.1 with only the nodes listed for which the state is stored.

9.1.2 Forking

POSIX based operating systems (like UNIX and Linux) allow a process to duplicate itself using the fork system call [10]. When a process calls this system call the process, including its state, will duplicate.
Linux implements this fork system call using copy-on-write. This means that not all memory pages have to be duplicated, but only the changes a child process makes need to be stored. This has two great benefits. Firstly it improves the speed of forking a process: in just a few milliseconds the forking can be done. Secondly it reduces the memory usage enormously, instead of needing many megabytes only a few kilobytes can be enough for an additional forked process. Thus many more forked processes will fit in the same amount of RAM.



```
SULOUTPUT (line): OFRAME_1_0_0_0
output:OFRAME_1_0_0_0
1):OFRAME_1_0_0_0

_0_0_0
1):OFRAME_1_0_0_0

0_0_1_0
1):OFRAME_0_0_1_0

0_0_1_0
1):OFRAME_0_0_1_0

0_1_0_0
1):OFRAME_0_1_0_0
```

Figure 15: Screenshot of VirtualBox after learning a model of the Bounded Retransmission Protocol.

We could use this functionality instead of DMTCP or VirtualBox to improve the learning speed. Instead of taking a checkpoint, nothing has to be done, apart from adding the process id of the current instance of the SUL to the result trie and sending a pause signal to the current instance of the SUL.

Instead of restoring a checkpoint we apply algorithm 3.

Algorithm 3 Restoring a forked instance

```
Send a resume signal to the instance which should be restored
Send a signal to notify the instance that it should fork itself
Read the process id of the new forked instance from StdOut (now is known that the forking is done)
Send a pause signal to the (old) instance
Send the desired inputs to the new forked instance
```

Note that using this approach all queries are saved (the corresponding processes remain open) to prevent that an additional step needs to be taken to stop the corresponding process. This is in contradiction to the checkpoint techniques using DMTCP or VirtualBox, where it takes additional time to store a query.

This requires injecting some code in the SUL. We need to implement a handler that forks the process when a specific user defined POSIX signal is received. In addition, we need to implement the communication to and from the process. The output is done using StdOut. However, child processes share their StdOut file descriptor with their parent process. To identify the output of each individual process, we add the process id to every print statement. To send inputs to an individual process, other unused user defined POSIX signals are used, which could be sent to a specific process.

Although forking is in principle an efficient technique, we ran into several issues when trying to implement this technique in practice. Unfortunately it doesn't work if the SUL is multi-threaded and it might also give problems with open file handlers, which can cause a crash of the whole SUL after a fork operation. So it cannot be applied in most real-life situations. Ideally forking could be combined with virtualization technology (such as VirtualBox or QEMU) by forking instances of virtual machines so that it works with just about any software and is also very fast, but we didn't manage to get that working due to these reasons. Nevertheless, we managed to implement this forking technique and learn the same simple model from the Bounded Retransmission protocol as we did with VirtualBox in section 9.1.1. In this case it gives a much greater speed improvement than DMTCP.

	baseline	with Forking
#MQ	545	545
Time MQ (s)	4020	2111
#EQ	504	504
Time EQ (s)	3913	443
Speed compared to baseline	1.00	3.11

9.1.3 DMTCP combined with QEMU

The virtualization software QEMU is an alternative to VirtualBox. It could run a SUL within a virtual Windows,Linux,BSD,etc environment. Therefore it is a generic solution. QEMU could be used in combination with DMTCP. Instead of using the built-in support of the virtualization software, DMTCP is used to create and restore checkpoints. According to [34] creating a snapshot (even including a guest filesystem and 1GB of allocated RAM) could be done within 300ms and restarting a checkpoint could be done in about 100ms. This is a order of magnitude faster than we have seen in our experiments in section 9.1.1 with the built-in snapshot support of VirtualBox. Therefore this approach would be ideal to have a solution that is both generic and fast. However, we unfortunately did not manage to reproduce the claimed store and restart times. So future research is needed to successfully implement this approach.

9.1.4 CRIU and Docker

CRIU (Checkpoint/Restore In Userspace) [3] is a direct alternative for DMTCP. Just like DMTCP it is able checkpoint Linux applications and it works fully in userspace. Contrary to DMTCP checkpoints are not transferable to another machine.

Docker [5] is a tool that can create software containers in which Linux applications can run, without the overhead of full virtualization software like VirtualBox. These containers are transferable to other machines. Docker on its own does not have the functionality to save the RAM state of the running applications within the container.

Both CRIU and Docker can be combined (see [9] and [6]) to have both the ability to checkpoint running Linux applications and to have transferable checkpoints.

Future research is needed to evaluate whether CRIU, possibly combined with Docker, provide better results (less overhead, smaller checkpoints) than DMTCP in speeding up LearnLib.

9.2 Alternative improvement techniques

In this thesis we proposed two improvement techniques, parallelization, applying checkpointing and the combination of both. Another technique can be the reduction of the waiting time for already known quiescence. Sometimes parts of the query have been executed before and therefore the outcome is already known. To detect quiescence (the absence of an output) LearnLib normally has to wait a certain time. If we know in advance that from a particular state a certain input will produce quiescence, we can reduce the waiting time. This technique is easy to implement. It can work in combination with parallelization. It could also be used in combination with checkpointing, except when all queries are saved, since in that case there will be no execution of parts of which the outcome is already known.
Nevertheless this technique is tricky, because there should still be enough

time for the SUL to go internally to the next state. If only a single transition would take longer than the waiting time, even during the execution of thousands of inputs, LearnLib probably will crash or the learned model will be incorrect. Therefore we decided not to use this technique in this thesis.

9.3 Alternative equivalence oracles

In this thesis we have only used as the random words equivalence oracle. But our proposed improvement techniques can also work with other equivalence oracles, such as the *W-method* [32], the *Wp-method* [33] or novel algorithms based on *distinguishing strings* [43] or a *user log metric* [47].

Checkpointing can be used with such alternative equivalence oracles without modification of any source code. Parallelization requires some modifications to the code of the equivalence oracle. The queries should be divided in sets that can be executed in parallel and the execution of all parallel instances should end when a counter example is found.

10 Future research

10.1 Compression

As mentioned earlier, we would like to have as tiny checkpoints as possible. This would lead to faster saving and restoring of states (in many cases the disk transfer rate is the bottleneck) and also to less required disk space.
Using compression we could reduce the size of checkpoints. DMTCP has built-in support for gzip, but compressing and decompressing the checkpoints requires so much CPU time it only reduces the execution speed. As an alternative a much faster compression algorithm could be implemented such as Snappy[14] or QuickLZ[12].

To get even better compression, the fact that most checkpoints are very similar could be used. For example a delta compression technique could be implemented in which only the differences between the checkpoint to be made and the initial checkpoint are stored (This initial checkpoint is then ideally stored on a very fast storage device such as a RAM disk).

A quick test using Xdelta3 [19], which is an implementation of such a delta compression technique, shows that huge compression ratios of above 99.9% are possible.

However there still will be a trade-off between the extra CPU time the compression and decompression takes and the benefits of having smaller checkpoints. Compression could be done in parallel to improve the compression speed, however, it is likely to be more efficient to spend the parallel CPU time on the execution of multiple instances of the SUL.

10.2 Deleting saved states

Suppose there is a node r in the result trie 4.1 that has a corresponding checkpoint. If at a certain point all possible direct child nodes also have a checkpoint, then we are sure that the checkpoint of node r will not be used any more. This is because the algorithm 2 that selects which checkpoint should be reused during the execution of new query always selects the checkpoint for which the longest possible part of the query has already been executed. At the time it is known that a certain checkpoint will not be reused any more, the checkpoint can be deleted to save disk space. The deleting operation will just take additional time, which is not useful, but if the used disk to store the checkpoints is relatively small, the regained disk space can be useful to save additional checkpoints on the same disk.

10.3 Saving states of partially executed queries

We have only looked at the saving states after executing complete membership queries in this research. However in some cases it might be useful to save also some states after the partial execution of queries. Many of the executed membership queries are extensions of previous queries, i.e. the partially executed part has already been a complete previously executed query. However, that is not the case with equivalence queries, those randomly generated queries are not based upon each other, so a partially executed equivalence query might not have been executed before. Since a partial executed query is shorter than the complete query, the chance that it can be reused will be larger. Therefore it might be beneficial to save the state of some of those partial executed equivalence queries.

10.4 Just-In-Time determination of saving checkpoints

We have seen in section 2.2 and 3.1 that it is likely that many membership queries have to be executed at the same time. Each time there is a set of membership queries which can be executed independently of each other. Instead of determining in advance which membership queries should be saved, it could also be determined just-in-time at the moment that the queries of a set are known. At that moment a strategy can be used that only saves the states of those queries which can be reused as much as possible within that set.
For example, see the following set of five queries

query	potential reuse count
start button	4
start button → coffee button	0
start button → start button	2
start button → start button → coffee button	0
start button → start button → start button	0

For each query of this set the potential reuse count within the set is calculated. A possible strategy is to save only those queries which have a potential reuse count higher than a certain threshold value. If for example a threshold value of 3 is chosen only the state after executing the first query will be saved.

Of course we don't know which queries can be reused during subsequently sets. Thus this strategy may not always select the overall best checkpoints to be saved, but it can guarantee that checkpoints are reused at least a certain amount of times (the specified threshold value). So there will not be wasted any more time on saving checkpoint that are never going to be reused.

This strategy could be extended by the knowledge of equivalence queries. With the used random words algorithm, there could be determined in advance which queries will be used (by generating the random queries at the start of the learning process; before any membership/equivalence query has been executed by the SUL). This will improve the accuracy of the overall estimated reuse count and contribute to selecting the overall best checkpoints to be taken.

Note that applying parallelism in combination with this strategy might not be ideal, because if parallelism is applied the order in which queries are executed can vary.

10.5 Prefetching and non-blocking saving of states

In the current situation the process of executing a query, saving the corresponding state and restoring a state and executing the remaining part of the next query is executed sequentially. Two optimizations can be made: prefetching and non-blocking saving of states.

We have seen in section 2.2 and 3.1 that it is likely that many membership queries have to be executed at the same time. Thus, in the case of membership queries, there is known shortly in advance that a query will be executed. In section 2.3 we have seen that all equivalence queries are generated in advance, so they are also known in advance. With the concept of prefetching this information could be used. The principle of prefetching is shown in figure 16(b). Instead of loading a checkpoint at the time it is needed, the checkpoint could already be loaded during the execution of the previous query.

The principle of non-blocking saving of states is shown in figure 16(c). The next query could already be executed during the saving of a checkpoint, unless the next query requires the checkpoint that is made at the same time.

Both prefetching and non-blocking saving of states could be combined as shown in figure 16(d).

This is once more a kind of parallelism. But since it is likely that saving and restoring of states are mainly disk intensive, and the SUL is mainly CPU intensive, we expect that the saving and restoring of checkpoints would not hinder the parallel execution of the SUL much.

11 Conclusion

In this thesis we researched how the execution speed of LearnLib could be improved. We have elaborated extensively two techniques, namely parallelization and a novel approach based on checkpoints. We have seen of both techniques that they can be beneficial to speedup LearnLib in theory. We have chosen the relatively new, but promising tool DMTCP to implement the concept of checkpointing in practice. It turned out that also in practice, using three on real life based software implementations, both techniques can be very successful in improving the speed. Parallelization is almost always very beneficial. Checkpointing using DMTCP can be very beneficial when

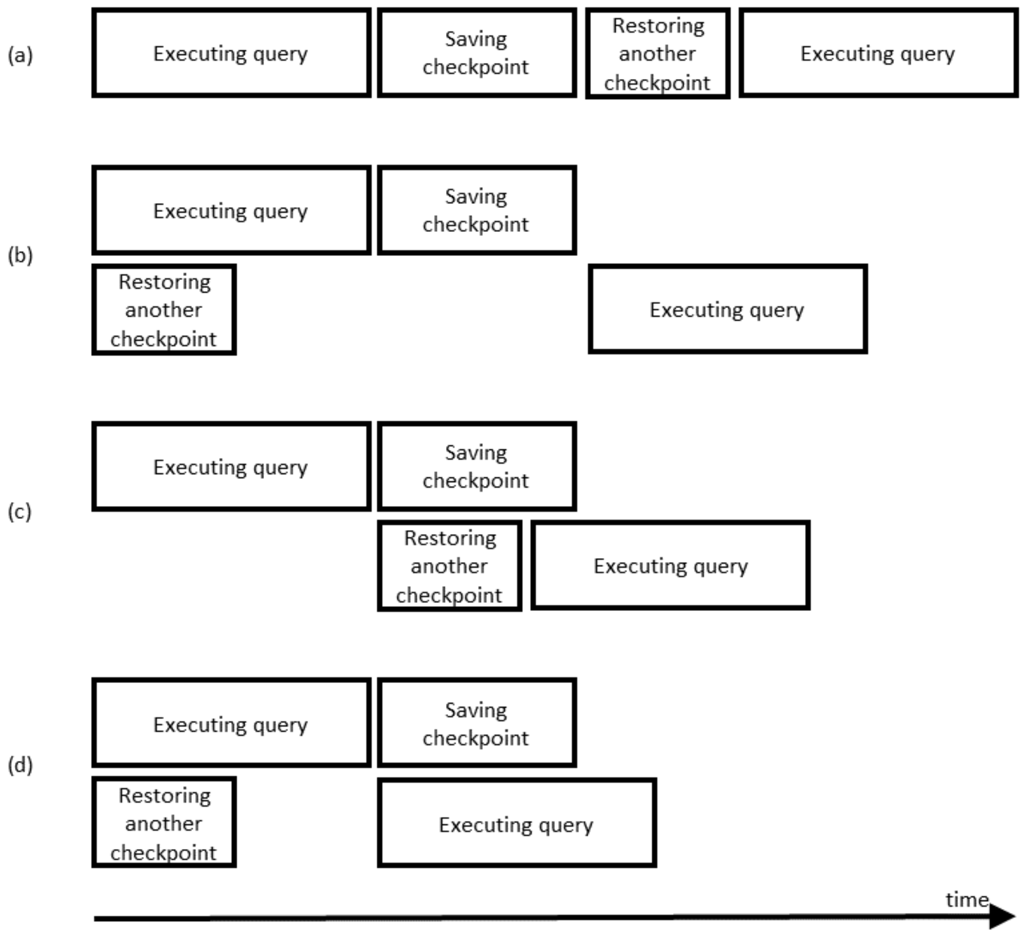


Figure 16: (a) Standard situation (b) Prefetching applied (c) Non-blocking saving of states applied (d) both prefetching and non-blocking saving of states applied

applied on a slowly responding SUL, however if applied on a fast responding SUL or used with a wrongly selected strategy to reduce the number of checkpoints it can slow down the learning process.
The highest speedups were achieved by combining parallelization with checkpointing, by which we achieved speed improvements of even up to about 300 times.
In addition, we have provided several ideas for improvements that could be worked out in future research, so hopefully even greater speed improvements are made possible in the future.

References

[1] Bart's Preinstalled Environment (BartPE) bootable live windows cd/dvd http://www.nu2.nu/pebuilder/.

[2] Client Hyper-V http://go.microsoft.com/fwlink/p/?LinkId=298966.

[3] Criu - checkpoint/restore in userspace http://www.criu.org/.

[4] DMTCP frequently asked questions http://dmtcp.sourceforge.net/FAQ.html.

[5] Docker - build, ship, and run any app, anywhere https://www.docker.com/.

[6] How to checkpoint and restore a docker container. http://criu.org/Docker.

[7] JamVM - a compact java virtual machine http://jamvm.sourceforge.net/.

[8] Javascript implementation of tic-tac-toe - stephen ostermiller - http://ostermiller.org/calc/tictactoe.html.

[9] Kubernetes blog - how did quake demo from dockercon work http://blog.kubernetes.io/2015/07/how-did-quake-demo-from-dockercon-work.html.

[10] Linux programmerś manual - fork(2) http://man7.org/linux/man-pages/man2/fork.2.html.

[11] Open-source learnlib: a from-scratch re-implementation of the former closed-source version http://learnlib.de/.

[12] QuickLZ - fast compression library for C, C# and Java http://www.quicklz.com/.

[13] Selenium - web browser automation http://www.seleniumhq.org/.

[14] Snappy - a fast compressor/decompressor https://github.com/google/snappy.

[15] Socat - multipurpose relay http://www.dest-unreach.org/socat/.

[16] Software bug contributed to blackout `http://www.securityfocus.com/news/8016`.

[17] VirtualBox `https://www.virtualbox.org/`.

[18] VMware products `http://www.vmware.com/products/`.

[19] xdelta - open-source binary diff, differential compression tools, vcdiff (rfc 3284) delta compression `http://xdelta.org/`.

[20] FD Aarts. Tomte: bridging the gap between active learning and real-world systems. 2014.

[21] Fides Aarts, Joeri De Ruiter, and Erik Poll. Formal models of bank cards for free. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 461–468. IEEE, 2013.

[22] Fides Aarts, Falk Howar, Harco Kuppens, and Frits Vaandrager. Algorithms for inferring register automata. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 202–219. Springer, 2014.

[23] Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In *Testing Software and Systems*, pages 188–204. Springer, 2010.

[24] Fides Aarts, Harco Kuppens, Jan Tretmans, Frits W Vaandrager, and Sicco Verwer. Learning and testing the bounded retransmission protocol. In *ICGI*, volume 21, pages 4–18. Citeseer, 2012.

[25] Fides Aarts, Julien Schmaltz, and Frits Vaandrager. Inference and abstraction of the biometric passport. In *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 673–686. Springer, 2010.

[26] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[27] Jason Ansel, Kapil Arya, and Gene Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[28] Arstechnica. Report: Airbus transport crash caused by wipe of critical engine control data `http://arstechnica.com/information-technology/2015/06/report-airbus-transport-crash`, 2015.

[29] Oliver Bauer, Johannes Neubauer, Bernhard Steffen, and Falk Howar. Reusing system states by active learning algorithms. In *Eternal Systems*, pages 61–78. Springer, 2012.

[30] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Hakansson, Paul Petterson, Wang Yi, and Martijn Hendriks. Uppaal 4.0. In *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, pages 125–126. IEEE, 2006.

[31] Axel Belinfante. Jtorx: A tool for on-line model-driven test derivation and execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–270. Springer, 2010.

[32] TS Chow. Testing software design modeled by finite-state machines.(1978). *IEEE Trans, on Soft. Engi*, pages 178–187.

[33] Susumu Fujiwara, Gregor V Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *Software Engineering, IEEE Transactions on*, 17(6):591–603, 1991.

[34] Rohan Garg, Komal Sodha, and Gene Cooperman. A generic checkpoint-restart mechanism for virtual machines. *arXiv preprint arXiv:1212.1787*, 2012.

[35] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[36] Ramon Janssen, Frits W Vaandrager, and Sicco Verwer. Learning a state diagram of tcp using abstraction. *Bachelor thesis, ICIS, Radboud University Nijmegen*, page 12, 2013.

[37] Phil Koopman. A case study of toyota unintended acceleration and software safety. *Presentation. Sept*, 2014.

[38] Watcharin Leungwattanakit, Cyrille Artho, Masami Hagiya, Yoshinori Tanabe, and Mitsuharu Yamamoto. Model checking distributed systems by combining caching and process checkpointing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 103–112. IEEE Computer Society, 2011.

[39] James D McCaffrey. *Software Testing: Fundamental Principles and Essential Knowledge.* BookSurge Publishing, 2009.

[40] Oliver Niese. *An integrated approach to testing complex systems.* PhD thesis, Universität Dortmund, 2003.

[41] Harald Raffelt, Bernhard Steffen, and Therese Berg. Learnlib: A library for automata learning and experimentation. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 62–71. ACM, 2005.

[42] Wouter Smeenk. Applying automata learning to complex industrial software. *Master's Thesis, Radboud University Nijmegen*, 2012.

[43] Rick Smetsers, Michele Volpato, Frits Vaandrager, and Sicco Verwer. Bigger is not always better: on the quality of hypotheses in active automata learning. In *The 12th International Conference on Grammatical Inference*, pages 167–181, 2014.

[44] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In *Formal Methods for Eternal Networked Software Systems*, pages 256–296. Springer, 2011.

[45] Max Tijssen, Erik Poll, and Joeri de Ruiter. Automatic modeling of SSH implementations with state machine learning algorithms. *Bachelor thesis, ICIS, Radboud University Nijmegen (June 2014)*, 2014.

[46] The New York Times. F.A.A. Orders fix for possible power loss in boeing 787 `http://www.nytimes.com/2015/05/01/business/faa-orders-fix-for-possible-power-loss-in-boeing-787.html`, 2015.

[47] Petra Van den Bos. Enhancing active automata learning by a user log based metric. *Master's Thesis, Radboud University Nijmegen*, 2015.

[48] Ana-Maria Visan, Kapil Arya, Gene Cooperman, and Tyler Denniston. Urdb: a universal reversible debugger based on decomposing debugging histories. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, page 8. ACM, 2011.

Appendix

A Comprehensive measurement results

A.1 Baseline

Bounded Retransmission:

#MQ	27215
Time MQ (s)	2739
#EQ	10410
Time EQ (s)	1536

Tic-Tac-Toe:

#MQ	20253
Time MQ (s)	17069
#EQ	5590
Time EQ (s)	6634

Engine State Machine:

#MQ	4848
Time MQ (s)	3864
#EQ	2580
Time EQ (s)	2546

A.2 Parallelization

Bounded Retransmission:

# parallel instances:	2	4	8	16	32	60
#MQ	27215	27215	27375	27215	27375	27055
Time MQ (s)	1520	812	443	245	123	66
#EQ	10823	11653	12536	11613	13539	15190
Time EQ (s)	796	430	233	108	68	43
Speed compared to baseline	1.85	3.44	6.32	12.11	22.38	39.22

Tic-Tac-Toe:

# parallel instances:	2	4	8	16	32	60
#MQ	19785	22567	21937	22819	20487	20721
Time MQ (s)	9115	5630	2690	1453	660	375
#EQ	5412	5700	5553	6157	6189	9275
Time EQ (s)	3313	1752	881	492	266	249
Speed compared to baseline	1.91	3.21	6.64	12.19	25.60	37.99

Engine State Machine:

# parallel instances:	2	4	8	16	32	60
#MQ	3971	4115	4187	1990	6313	4302
Time MQ (s)	1900	1046	577	125	244	88
#EQ	2093	2174	2435	2071	4370	6463
Time EQ (s)	1033	542	306	136	179	167
Speed compared to baseline	2.19	4.04	7.26	24.56	15.15	25.14

A.3 DMTCP

A.3.1 Strategy 1: limiting checkpoints using a linear function

Bounded Retransmission:

query size that is a multiple of:	1	2	3	4	5	10	20
#MQ	27215	27215	27215	27215	27215	27215	27215
Time MQ (s)	3174	2224	2046	2134	2107	2656	3089
#EQ	10410	10410	10410	10410	10410	10410	10410
Time EQ (s)	1401	1440	1432	1628	1686	1685	1684
#checkpoints	24594	12008	8130	6717	4734	2503	30
Speed compared to baseline	0.93	1.17	1.23	1.14	1.13	0.98	0.90

Tic-Tac-Toe:

query size that is a multiple of:	1	2	3	4	5	10	20
#MQ	20253	20253	20253	20253	20253	20253	20253
Time MQ (s)	13672	11721	9613	10730	19681	18229	18771
#EQ	5590	5590	5590	5590	5590	5590	5590
Time EQ (s)	3878	4973	4366	5161	7325	7084	7225
#checkpoints	18142	8071	4532	4370	8179	313	1
Speed compared to baseline	1.35	1.42	1.70	1.49	0.88	0.94	0.91

Engine State Machine:

query size that is a multiple of:	1	2	3	4	5	10	20
#MQ	4848	4848	4848	4848	4848	4848	4848
Time MQ (s)	1635	1399	1545	1876	2170	2137	2254
#EQ	2580	2580	2580	2580	2580	2580	2580
Time EQ (s)	1140	1190	1296	1434	1748	1525	1528
#checkpoints	3998	1956	1346	996	870	296	7
Speed compared to baseline	2.31	2.48	2.26	1.94	1.64	1.75	1.69

A.3.2 Strategy 2: limiting checkpoints using an upper limit

Bounded Retransmission:

query size that is max:	1	2	3	4	5	10	20
#MQ	27215	27215	27215	27215	27215	27215	27215
Time MQ (s)	2939	2664	2500	2422	2287	2719	3174
#EQ	10410	10410	10410	10410	10410	10410	10410
Time EQ (s)	1590	1509	1419	1383	1372	1395	1398
#checkpoints	11	111	912	2206	3881	16732	24584
Speed compared to baseline	0.94	1.02	1.09	1.12	1.17	1.04	0.94

Tic-Tac-Toe:

query size that is max:	1	2	3	4	5	10	20
#MQ	20253	20253	20253	20253	20253	20253	20253
Time MQ (s)	15499	11654	9357	9046	11844	14827	13672
#EQ	5590	5590	5590	5590	5590	5590	5590
Time EQ (s)	6085	5150	4483	4167	5637	4244	3878
#checkpoints	10	91	820	4563	12402	17015	18142
Speed compared to baseline	1.10	1.41	1.71	1.79	1.36	1.24	1.35

Engine State Machine:

query size that is max:	1	2	3	4	5	10	20
#MQ	4848	4848	4848	4848	4848	4848	4848
Time MQ (s)	2463	1939	1757	1764	1786	1817	1635
#EQ	2580	2580	2580	2580	2580	2580	2580
Time EQ (s)	1652	1310	1209	1172	1194	1171	1140
#checkpoints	7	43	154	407	774	2514	3998
Speed compared to baseline	1.56	1.97	2.16	2.18	2.15	2.15	2.31

A.3.3 Strategy 3: limiting checkpoints using a lower limit

Bounded Retransmission:

query size that is min:	1	2	3	4	5	10	20
#MQ	27215	27215	27215	27215	27215	27215	27215
Time MQ (s)	3245	3272	3339	3496	4091	3027	3146
#EQ	10410	10410	10410	10410	10410	10410	10410
Time EQ (s)	1395	1419	1474	2385	1735	1705	1710
#checkpoints	24594	24584	24484	23683	22389	10336	40
Speed compared to baseline	0.92	0.91	0.89	0.73	0.73	0.90	0.88

Tic-Tac-Toe:

query size that is min:	1	2	3	4	5	10	20
#MQ	20253	20253	20253	20253	20253	20253	20253
Time MQ (s)	13672	13070	13690	15097	20965	17956	18771
#EQ	5590	5590	5590	5590	5590	5590	5590
Time EQ (s)	3878	3899	4015	5051	7767	7253	7225
#checkpoints	18142	18133	18052	17323	13580	1440	1
Speed compared to baseline	1.35	1.40	1.34	1.18	0.82	0.94	0.91

Engine State Machine:

query size that is min:	1	2	3	4	5	10	20
#MQ	4848	4848	4848	4848	4848	4848	4848
Time MQ (s)	1635	1730	1632	1745	1764	1891	2254
#EQ	2580	2580	2580	2580	2580	2580	2580
Time EQ (s)	1140	1186	1281	1403	1502	1523	1528
#checkpoints	3998	3992	3956	3845	3592	1774	7
Speed compared to baseline	2.31	2.20	2.20	2.04	1.96	1.88	1.69

A.3.4 Strategy 4: limiting checkpoints using an exponential function

Bounded Retransmission:

#MQ	27215
Time MQ (s)	2114
#EQ	10410
Time EQ (s)	1479
#checkpoints	4784
Speed compared to baseline	1.19

Tic-Tac-Toe:

#MQ	20253
Time MQ (s)	9401
#EQ	5590
Time EQ (s)	4582
#checkpoints	4146
Speed compared to baseline	1.70

Engine State Machine:

#MQ	4848
Time MQ (s)	1760
#EQ	2580
Time EQ (s)	1235
#checkpoints	769
Speed compared to baseline	2.14

A.4 Combining parallelism with checkpointing

Here 60 parallel SULs are combined with checkpointing using the exponential function (strategy 4).

Bounded Retransmission:

#MQ	27055
Time MQ (s)	185
#EQ	15197
Time EQ (s)	104
#checkpoints	4372
Speed compared to baseline	14.79

Tic-Tac-Toe:

#MQ	22945
Time MQ (s)	376
#EQ	9785
Time EQ (s)	205
#checkpoints	4430
Speed compared to baseline	40.80

Engine State Machine:

#MQ	4380
Time MQ (s)	59
#EQ	6784
Time EQ (s)	93
#checkpoints	601
Speed compared to baseline	42.17

A.5 Slowing down the SULs

Bounded Retransmission:
The bounded Retransmission Protocol simulation is slowed down so that a reset takes 2000ms (instead of 20ms) and a step takes 20ms (instead of 10ms):

	baseline	DMTCP	parallelism	both combined
#MQ	27215	27215	26895	27055
Time MQ (s)	53638	2611	967	181
#EQ	10410	10410	15187	14990
Time EQ (s)	23446	2382	642	78
#checkpoints	N/A	4784	N/A	4372
Speed compared to baseline	1.00	15.44	47.91	297.62

Tic-Tac-Toe:
The Tic-Tac-Toe game is slowed down so that a reset takes 5000ms (instead of 226ms) and a step takes 500ms (instead of 163ms):

	baseline	DMTCP	parallelism	both combined
#MQ	20253	20253	22945	22945
Time MQ (s)	142236	20646	3484	1064
#EQ	5590	5590	9259	10208
Time EQ (s)	58688	22145	2274	993
#checkpoints	N/A	4146	N/A	4429
Speed compared to baseline	1.00	4.70	34.89	97.68

Engine State Machine:
The Engine State Machine simulation is slowed down so that a reset takes 2000ms (instead of 500ms) and a step takes 100ms (instead of 50ms):

	baseline	DMTCP	parallelism	both combined
#MQ	4848	4848	4380	4380
Time MQ (s)	11720	2813	254	93
#EQ	2580	2580	6469	7083
Time EQ (s)	7620	2088	499	153
#checkpoints	N/A	769	N/A	596
Speed compared to baseline	1.00	3.95	25.68	78.62