Radboud University Nijmegen

Master Thesis Computer Science

Applying Automata Learning to Complex Industrial Software

by Wouter Smeenk
woutersmeenk@gmail.com

Date: 12th September 2012
Thesis number: 661

Supervisors:
Prof. Frits Vaandrager
(Radboud University Nijmegen)
Dr. Lou Somers
(Océ)

Abstract

Learning models of machines comes very naturally to humans. They construct a mental model of the behaviour of machines while trying out different options. For machines this is much harder. Algorithms have been developed to learn the behaviour of systems and describe them in state machine models. Once such a model has been learned it can be used by software engineers to improve their software. They can simulate and analyse the behaviour of the system, test newer versions of the system and get insight in legacy systems of which no documentation exists.

This master thesis describes a case study done at Océ for the ITALIA project at the Radboud University Nijmegen. The goal of this research was to find out whether state-of-the-art techniques and tools for automata learning are powerful enough to learn models of industrial control software. Specifically whether the LearnLib tool developed at the University of Dortmund was able to learn a model of the Engine Status Manager (ESM) developed at Océ.

The ESM controls the transition from one status to another in a printer. It delegates status requests to the connected hardware components and coordinates their responses. Software like the ESM can be found in many embedded systems in one form or another. Although such the ESM may seem simple, the many details and exceptions involved make it hard to learn.

A system is learned in two alternating phases: A learning phase in which a hypothesis model is created, and a testing phase in which the algorithm searches for a counterexample. This is implemented in the LearnLib. In order to find counterexamples during the test phase using less queries novel techniques had to be developed.

Although a correct model of the ESM could not be learnt during this research novel techniques were developed that greatly reduce the number of test queries needed.

De schrijver werd door Océ-Technologies B.V. in staat gesteld een onderzoek te verrichten dat mede aan dit rapport ten grondslag ligt.

Océ-Technologies B.V. aanvaardt geen verantwoordelijkheid voor de juistheid van de in dit rapport vermelde gegevens, beschouwingen en conclusies, die geheel voor rekening van de schrijver komen.

Contents

1 Introduction

Model-based testing has become important in software engineering. Therefore many tools and theories have been developed, such as JTorx [Bel10] which is based on the ioco-testing theory by Tretmans [Tre08]. In order to use these tools a model of the *system under test (SUT)* is needed. This model is constructed manually in most cases but in software engineering projects the time required to create such a model is not always available. Another problem can be the lack of documentation and knowledge about the SUT.

In such a situation it is beneficial to automatically learn the behaviour of the SUT. In some cases it is feasible to do this in a systematic way. A widely used method is known as *automata learning* or *regular inference* developed by Angluin [Ang87]. This method has been further refined to learn Mealy machines by Niese [Nie03]. The tool LearnLib [RSBM09] implements this method.

In practice automata learning can be useful in various ways. The learned model of a software system can be used to test whether a new version of the software has the same behaviour (regression testing). A model of a reference implementation can be learned and an alternative implementation can be tested using the learned model. Learning models of legacy software of which no documentation or knowledge exists any more in order to test a new implementation of this software.

Using the current version of LearnLib it is only feasible to learn a SUT if it has a small input alphabet. Complex systems such as communication protocols have very large or infinite input alphabets. To solve this problem Aarts, Jonsson and Uijen [AJU10] proposed an abstraction mapping technique. Using this technique it is possible to learn more complex systems when suitable abstractions can be found.

This master thesis describes a case study done at Océ. The software system that was studied is called the *Engine Status Manager (ESM)*. The ESM is responsible for controlling the status of a printer. It was chosen because it is used in many Océ printers. The ESM is very stable and has been in use for 10 years. It is also well documented and an extensive test suite exists. This makes it easy to understand the ESM and its interfaces. The purpose of this case study is to investigate if it is feasible to learn typical software systems developed at Océ. The research is carried out in the context of the ITALIA project [ITA] at the Radboud University Nijmegen.

The ESM has also been studied in other research projects. Ploeger [Plo05]

modelled the ESM and other related managers and verified properties based on the official specifications of the ESM. Graaf and van Deursen [GvD07] have checked the consistency of the behavioural specifications defined in the ESM against the state chart definition of the ESM.

This research tries to answer the following research question:

Research question: Can automata learning successfully be applied to industrial control software?

Industrial software can be defined as software, specifically control software, developed in an industrial setting for example in aeroplanes, cars, satellites and printers.

To define the research question more clearly it is divided into the following subquestions:

- How can a model of the ESM be learned using LearnLib?
- Does the learned model conform to the specification?
- How can structure be inferred in the learned model?

The research described in this master thesis is divided in two parts. During the first part an interface between ESM and LearnLib is created which is used to learn the ESM. This answers the first subquestion. During the second part a model is created manually in order to verify the correctness of the learned model, which answers the second subquestion. It was quickly discovered that more time than available was required in order to answer the third subquestion. Although this subquestion was initially part of this master thesis it was not answered in detail.

Section 2 describes the ESM in detail. Section 3 describes the learning techniques used. Section 4 describes the first part of the research. Section 5 describes the first part of the research. Future research and the conclusion are discussed in section 6.

2 Engine Status Manager

The focus of this master thesis is the *Engine Status Manager (ESM)*. This manager is used within many Océ products to manage the status of the engine of the printer or copier. In this section the overall structure and context of the ESM will be explained.

2.1 ESRA

The requirements and behaviour of the ESM are defined in a software architecture called Embedded Software Reference Architecture (ESRA). The components defined in this architecture are reused in many of the products developed by Océ and form an important part of these products. This architecture is developed for *cut-sheet* printers or copiers. The term cut-sheet refers to the use of separate sheets of paper as opposed to a *continuous feed* of paper.

An *engine* refers to the printing or scanning part of a printer or copier. Other products can be connected to an engine that pre- or postprocesses the paper for example a cutter, folder, stacker or stapler.

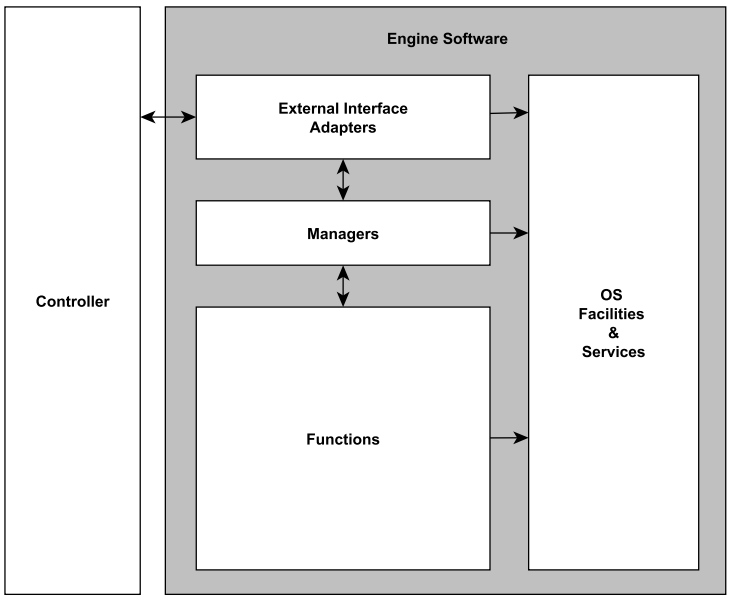

Figure 1: Global overview of the engine software

Figure 1 shows an overview of the software in a printer or copier. The *controller* communicates the required actions to the engine software. This includes transport of digital images, status control, print or scan actions and error handling. The controller is responsible for queueing, processing the actions received from the network and operators and delegating the appropriate actions to the engine software.

The *managers* communicate with the controller using the *external interface adapters*. These adapters translate the external protocols to internal protocols. The managers manage the different functions of the engine. They are divided by the different functionalities such as status control, print or scan actions or error handling they implement. In order to do this the manager may communicate with other managers and functions.

A *function* is responsible for a specific set of hardware components. The functions translate commands from the managers to the function hardware and report the status and other information of the function hardware to the managers. This hardware can for example be the printing hardware or hardware that is not part of the engine hardware such as a stapler.

Other functionalities such as logging and debugging are orthogonal to the functions and managers.

2.2 ESM and connected components

The ESM is responsible for the transition from one status of the printer or copier to another. It coordinates the functions to bring them in the correct state according to the requested status. Moreover, it informs all its connected clients (managers or the controller) of status changes. Finally, it handles status transitions when an error occurs.

Figure 2 shows the different connections to the ESM. The *Error Handling Manager (EHM)*, *Action Control Manager (ACM)* and other clients request engine statuses. The ESM decides whether a request can be honoured immediately, has to be postponed or ignored. If the requested action is processed the ESM requests the functions to go to the appropriate status. The EHM has the highest priority and its requests are processed first. The EHM can request the engine to go into the defect state. The ACM has the next highest priority. The ACM requests the engine to switch between running and standby mode. The other clients request transitions between the other states, such as idle, sleep, standby and low power. All the other clients have the same lowest priority.

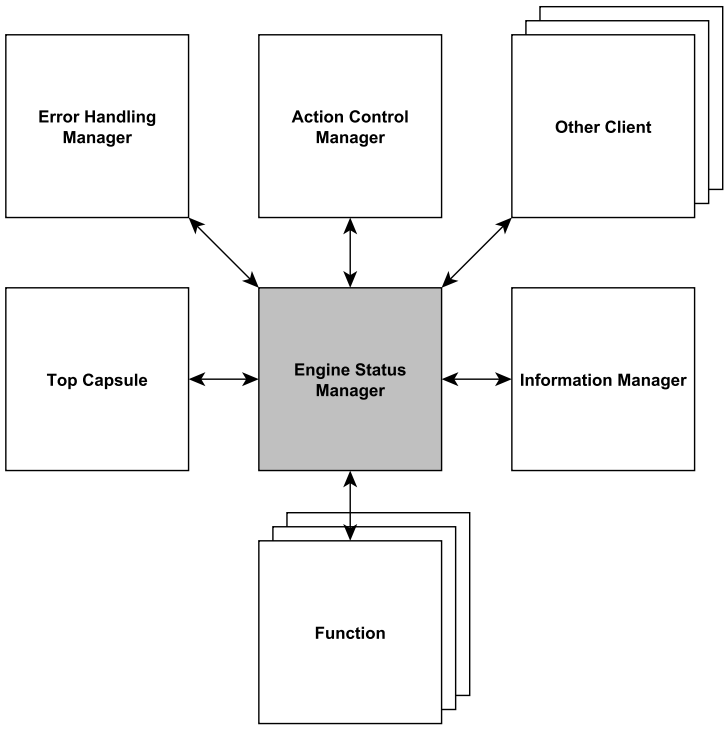

Figure 2: Overview of the managers and clients connected to the ESM

The Top Capsule instantiates the ESM and communicates with it during the initialisation of the ESM. The Information Manager provides some parameters during the initialisation.

There are more managers connected to the ESM but they are of less importance and are thus not mentioned here.

2.3 Rational Rose Real Time

The ESM has been implemented using *Rational Rose Real Time (RRRT)*. In this tool so-called *capsules* can be created. Each of these capsules defines a hierarchical *state diagram*. Capsules can be connected with each other using *structure diagrams*. Each capsule contains a number of ports that can be connected to ports of other capsules by adding connections in the associated structure diagram. Each of these ports specifies which protocol should be used. The protocol defines which messages may be send to and from the port. Transitions in the state diagram of the capsule can be triggered by

arriving messages on a port of the capsule. Messages can be sent to these ports using the action code of the transition. The transitions between the states actions and guards are defined in C++ code. From the state diagram, C++ source files are generated.

RRRT is heavily based on UML [Obj04]. Many of the semantics defined in UML are used in RRRT. One important semantic used in RRRT is the run-to-completion execution model. This means that when a received message is processed, the execution cannot be interrupted by other arriving messages. These messages are placed in a queue to be processed later.

2.4 The ESM state diagram

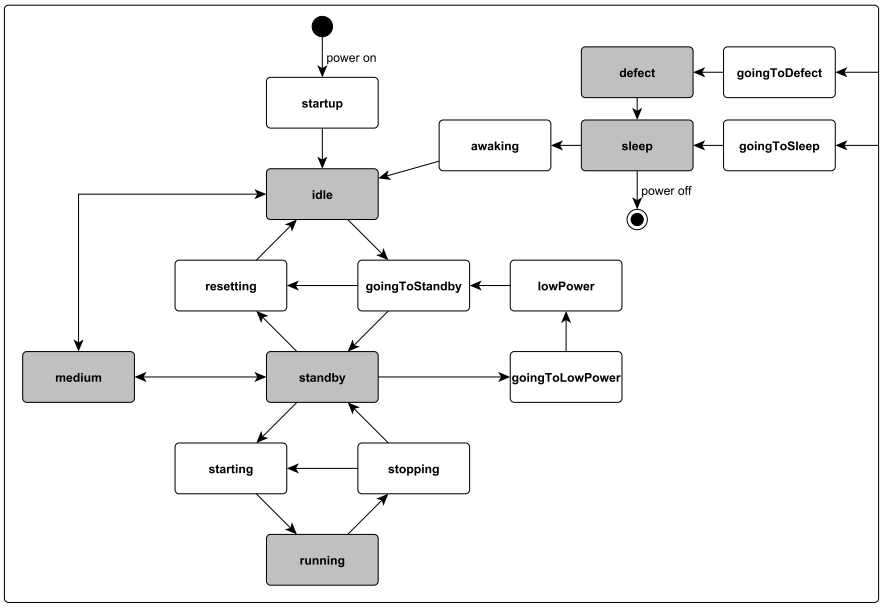

Figure 3: Top states and their transitions in the ESM

Figure 3 shows the top states of the ESM state diagram. The states that can be requested by the clients and managers are grey. The other states are so called *transitory states*. In transitory states the ESM is waiting for the functions to report that they have moved to the corresponding states. Once all functions have reported this the ESM moves to the corresponding state.

The `idle` state indicates that the engine is done starting up but that it is still *cold* (uncontrolled temperature). The `standby` state indicates that

the engine is *warm* and ready for printing or scanning. The `running` state indicates that the engine is printing or scanning.

The transition from the border to the `goingToSleep` and `goingToDefect` states indicates that it is possible to move to the `sleep` or `defect` state from any state. In some cases it is possible to awake from sleep mode in other cases the main power is turned off.

The `medium` state is designed for diagnostics. In this state the functions can each be in a different state. For example one function is in standby state while a other function is in idle state.

The state diagram in figure 3 may seem simple, but it hides many details. Each of the states has up to 5 nested states. In total there are 70 states that do not have further nested states. The C++ code contained in the actions of the transitions is in some cases non-trivial. The possibility to transition from any state to the sleep or defect state also complicates the learning.

3 Automata learning

This section will give background information on the learning techniques used in this research. The section 3.1 will introduce Mealy machines and their notations. Section 3.2 will introduce a technique to learn such a model from a SUT.

3.1 Mealy machines

A *Mealy machine* is defined as a tuple $M = \langle I, O, Q, q_0, \delta, \lambda \rangle$, where

- I is the finite non-empty set of input symbols also called the input alphabet.
- O is the finite non-empty set of output symbols also called the output alphabet.
- Q is the finite non-empty set of states.
- $q_0 \in Q$ is the initial state.
- $\delta : Q \times I \to Q$ is the transition function.
- $\lambda : Q \times I \to O$ is the output function.

An informal description of a Mealy machine can be given as follows. Given a Mealy machine in a state $q \in Q$ when an input $i \in I$ is processed an output $\lambda(q, i)$ will be given. The Mealy machine will then be in state $\delta(q, i)$. This means that any input into a Mealy machine will always produce an output. A Mealy machine defined in this way will always be *deterministic*. Only deterministic Mealy machines are considered in this research.

A *transition* of a Mealy machine is denoted as $q \xrightarrow{i/o} q'$ meaning $\delta(q, i) = q'$ and $\lambda(q, i) = o$. δ and λ can be extended to accept sequences of inputs instead of only single inputs. This is done by defining

- $\delta(q, \epsilon) = q$
- $\delta(q, pi) = \delta(\delta(q, p), i)$
- $\lambda(q, \epsilon) = \epsilon$
- $\lambda(q, pi) = \lambda(q, p)\lambda(\delta(q, p), i)$

where $p \in I^*$, $i \in I$ and $q \in Q$.

3.2 Learning algorithm

This section explains the automata learning algorithm used in this research is explained. This technique is also often called *regular inference*.

The algorithm described here is based on the L^* *algorithm* as first proposed by Dana Angluin [Ang87]. The L^* algorithm is used to infer *deterministic finite automata (DFA)*. Later it was adapted by Niese [Nie03] in order to also be able to learn Mealy machines. Our description of the adaptation of the L^* algorithm by Niese is based on the work of Bohlin [Boh09]. An overview of the these algorithms is given by Stefen, Hower and Merten [SHM11].

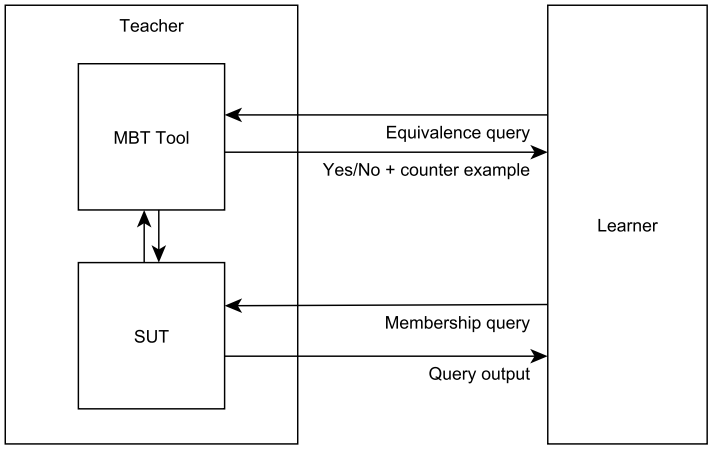

Figure 4: Learner overview

Figure 4 shows an overview of the algorithm. The *learner* can ask the *teacher* two types of queries: a *membership query* or an *equivalence query*. It is assumed the teacher has a SUT that can be described by a Mealy machine $M = \langle I, O, Q, q^0, \delta, \lambda \rangle$ and that it is *input complete*. A membership query by the learner is a sequence of input symbols from the alphabet I. The teacher responds with the query output, a sequence of output symbols from the alphabet O. The teacher can also perform a reset that returns the SUT to the initial state. When the learner asks an equivalence query it gives the teacher a *hypothesis model* of the SUT and asks if it is equivalent to the SUT. The hypothesis model is a Mealy machine that represents the best estimate the learner has about the SUT. If it is equivalent, the teacher responds with `yes`, otherwise it responds with `no` and gives a sequence of input symbols for which the SUT gives a different output than the hypothesis.

In practise the equivalence query is approximated by using a model-based testing (MBT) tool. This tool generates membership queries and tests if there is a difference in the output given by the SUT and the hypothesis. These queries are also called test queries. LearnLib implements both the learning algorithm and the MBT tool.

The learner begins by asking membership queries and constructing a hypothesis model from the outputs. It then asks an equivalence query. When the teacher responds with a counterexample the learner refines its hypothesis and tries again. When the teacher responds with `yes` the learning is done.

In order to create a hypothesis an *observation table* is used. An observation table is tuple $OT = (S, E, T)$. Here $S \subseteq I^*$ is the non-empty prefix closed finite set of prefixes, $E \subseteq I^*$ is the non-empty finite set of suffixes and T is a function which maps each cell of the table to an output sequence in O^*. Formally the type of T is defined as $T : ((S \bigcup S \cdot I) \times E) \rightarrow O^*$. For every $s \in S \bigcup S \cdot I$ and $e \in E$ we have $T(s, e) = \lambda(q_0, se)$. For every $s \in S \bigcup S \cdot I$ the function $row(s)$ is defined as $row(s)(e) = T(s, e)$ for every $e \in E$. The cells of T are filled using membership queries. A reset before each membership query is used to ensure the SUT is in the initial state.

The elements of $S \bigcup S \cdot I$ are the row labels of the table and elements of E are the column labels of the table. Informally one can say that each cell of the table is filled with the output of the Mealy machine when using the corresponding row label and column label concatenated together as input. Rows with the labels in the set S can be seen as the states of the Mealy machine. Rows with labels in the set $S \cdot I$ can be seen as the transitions of the Mealy machine.

The set S is initialized as $\{\epsilon\}$. The set E is initialized as I. In order to create a hypothesis model from the observation table it must be *closed* and *consistent*.

The observation table is closed if for each $s \in S \cdot I$ there exists an $s' \in S$ such that $row(s) = row(s')$. Informally one can say that if the observation table is not closed there is a transition from a state that leads to a unknown state. The observation table can be closed by adding a new prefix for the unknown state. The algorithm finds $s \in S$ and $a \in I$ such that $row(sa) \neq row(s'a)$ for all $s' \in S$ and adds sa to S and the new cells are filled using the corresponding membership queries.

The observation table is consistent if for all $s, s' \in S$, $row(s) = row(s')$ implies that $row(sa) = row(s'a)$ for all $a \in I$. Informally this means that if the observation table is inconsistent there are two states that are equal but

when a transition $a \in I$ is taken in each state they do not end up in the same state. The inconsistency can be removed by adding a new suffix. The algorithm finds $s, s' \in S$, $a \in I$ and $e \in E$ such that $row(s) = row(s')$ but $T(sa, e) \neq T(s'a, e)$ and adds the suffix ae to E.

When the observation table is closed and consistent a hypothesis $H = \langle I, O^H, Q^H, q_0^H, \delta^H, \lambda^H \rangle$ can be constructed as follows:

- O^H is the set of all the output symbols used in the observation table.
- $Q^H = \{row(s) | s \in S\}$
- $q_0^H = row(\epsilon)$
- $\delta^H(row(s), a) = row(sa)$
- $\lambda^H(row(s), a) = T(s, a)$

The hypothesis is then given to the teacher. If the hypothesis is incorrect a counterexample $w \in I^*$ will be give such that $\lambda(q_0, w) \neq \lambda^H(q_0^H, w)$. All the prefixes of w will then be added to S. The algorithm will extend the observation table until it is consistent and closed and create a new hypothesis. This will continue until the teacher declares the hypothesis correct.

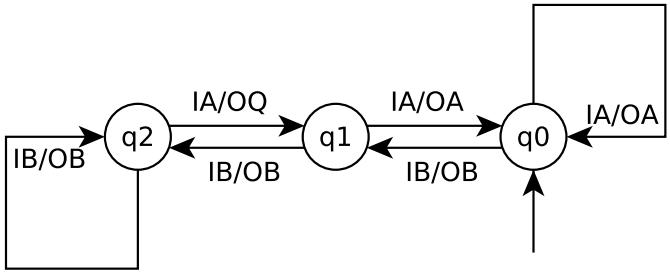

Figure 5: Example Mealy machine, with start state `q0`

In order to demonstrate the learning algorithm the Mealy machine in figure 5 will be used as SUT. In order to start learning S is initialized to $\{\epsilon\}$. The observation table is filled as shown in table 1. The rows of the prefixes in S are displayed in the top half. The prefixes in $S \cdot I$ are displayed in the bottom half.

This table is closed and consistent according to the above definitions. However the hypothesis constructed from this table only contains a single

OT_1	IA	IB
ϵ	OA	OB
IA	OA	OB
IB	OA	OB

Table 1: Observation table 1

state `q0`. The sequence `IB IB IA` is a counterexample. It results in the output `OA` in the hypothesis and in `OQ` in the SUT. All the prefixes of the sequence `IB IB IA` are added to S. The table is filled using the new prefixes of S as shown in 2.

OT_2	IA	IB
ϵ	OA	OB
IB	OA	OB
IB IB	OQ	OB
IB IB IA	OA	OB
IA	OA	OB
IB IA	OA	OB
IB IB IA IA	OA	OB
IB IB IA IB	OQ	OB
IB IB IB	OQ	OB

Table 2: Observation table 2

This table is closed, but it is not consistent. Row `IB` and row ϵ are equal but the rows `IB` and `IB IB` are not. The suffix `IA` gives a different output. We can thus add the suffix `IB IA` to resolve the inconsistency. This results in table 3. The hypothesis constructed from this table is equivalent to the SUT.

OT_3	IA	IB	IB IA
ϵ	OA	OB	OA
IB	OA	OB	OQ
IB IB	OQ	OB	OQ
IB IB IA	OA	OB	OQ
IA	OA	OB	OA
IB IA	OA	OB	OA
IB IB IA IA	OA	OB	OA
IB IB IA IB	OQ	OB	OQ
IB IB IB	OQ	OB	OQ

Table 3: Observation table 3

4 Learning the ESM

4.1 Learning set-up

In order to learn the behaviour of the ESM it was connected to the LearnLib tool [MSHM11]. LearnLib is developed at the University of Dortmund and implements the learning algorithms described in section 3.2.

LearnLib needs to send an input query and receive the output sequence. Before each input query is sent, the SUT needs to be in its initial state. This means that LearnLib needs a way to reset the SUT.

A clear interface to the ESM has been defined in RRRT. The ESM defines ports from which it receives a predefined set of inputs and to which it can send a predefined set of outputs. However this interface can only be used within RRRT. In order to communicate with the LearnLib software a TCP connection is used. An extra capsule is created in RRRT which connects to the ports defined by the ESM. This capsule creates a TCP connection to LearnLib, inputs and outputs are translated to and from a string format and sent over the connection.

The inputs and outputs sent to and from the ESM have parameters. The parameters are enumerations (of statuses) or integers bounded by the number of functions connected to the ESM. Currently LearnLib cannot handle inputs with parameters. One input of each combination of input and parameter is generated for the input alphabet used by LearnLib.

In order to reduce the number of inputs some are grouped together. When learning the ESM using 1 function 83 inputs can be grouped. When using 2 functions 126 inputs can be grouped. A new input input I is added to the alphabet to represent a group of inputs. When I needs to be sent to the ESM, one input of the represented group is randomly selected. This is a valid abstraction because all the inputs in the group have exactly the same behaviour in any state of the ESM. This has been verified by doing code inspection. No other abstractions were found during the research. After the inputs are grouped a total of 82 inputs remain when learning the ESM using 1 function. 105 inputs remain when using 2 functions.

The ESM also cannot be modelled by a directly Mealy machine. Some inputs may produce no or more than one output. In order to compensate for this we first tried to use a technique described by Aarts and Vaandrager [AV10] was used. A learning purpose, that fits the run-to-completion execution model used by RRRT, was used. This means that no new input is sent be-

fore the SUT is quiescent. The learning purpose that fits this has two states. In the initial state only inputs may be sent or a quiescence output received. When an input is received the second state is reached and only outputs may be received. When a quiescence output is received the initial state is reached again.

This approach works well in general but as also noted by Aarts and Vaandrager [AV10] the technique may cause LearnLib to ask unnecessary queries. For each state where an output is expected all the inputs that are not allowed by the learning purpose are also tried. In order to optimize this the learning purpose is removed and a direct translation is made. When an input is sent all the outputs are collected until quiescence is detected. All the outputs are concatenated and viewed as a single output. This allows us to view the ESM as a Mealy machine. This is only possible because of the run-to-completion execution model used in RRRT.

Normally quiescence is detected by waiting for a fixed timeout period. However this causes the system to be mostly idle while waiting for the timeout, which is very inefficient. In order to detect quiescence immediately the run-to-completion execution model used by RRRT is exploited. The ESM was modified to respond to a special input with a single special output. This special input is sent after each input with a low priority. Only when the original input is processed and all the outputs are sent will the special input be processed and the special output be sent back.

4.2 Test selection strategies

The ESM was learned using the set-up described in section 4.1. In this case study the most challenging problem was finding counterexamples for the hypotheses constructed during learning.

In order to find these counterexamples, LearnLib implements several algorithms one of which is a random walk algorithm. The random walk algorithm works by first selecting the length of the test query. This is done by starting with the lower bound of the length. The length is increased using a while loop until it is equal to the upper bound or a random boolean variable evaluates to true. Each of the input symbols in the test query is then randomly selected from the input alphabet I from a uniform distribution.

Normally, LearnLib uses a fixed upper bound for the maximum number of test queries executed after the previous counterexample has been found. It was however quickly discovered that it is hard to judge what this upper

bound should be. At the beginning of the research LearnLib was left running without upper bound on the number of test queries. The human operator judged whether LearnLib should be terminated. The observations of the various runs during this research lead to the following heuristic: If n test queries are needed to find the last counterexample, then $n * 10$ test queries will be executed before terminating LearnLib.

In order to find counterexamples a specific sequence of input symbols is needed to arrive at the state in the SUT that differentiates it from the hypothesis. The upper bound for the size of this search space is $|I|^n$ where $|I|$ is the size of the alphabet used and n the length of the counterexample that needs to be found. If this sequence is long the chance of finding it is small. Because the ESM has many different input symbols to choose from, finding the correct one is hard. When learning the ESM with 1 function there are 82 possible input symbols. If for example the length of the counter example needs to be at least 6 inputs to identify a certain state. The upper bound on the number of test queries would then be $30,4 * 10^{11}$. An average test query takes 5 milliseconds, thus it would take about 48,2 years to execute these test queries.

In order to find all counterexamples needed to learn the correct model of the ESM, the standard random search used in LearnLib was altered. Four different techniques were added. These techniques greatly improve the expected time that is needed to find counterexamples. Each of the techniques will be explained in this section and empirical data will be presented in order to show the improvements.

4.2.1 Random prefix selection

The learned model of the ESM has many states that have a long access sequence. This can be seen in figure 6. Most access sequences have a length between 15 and 23 inputs and the average length is 18,22 inputs.

Using the random walk test selection, states with a shorter access sequence are visited more often than states with a long access sequence. Some counterexamples may only be found after visiting certain states. If these states have a lower chance of being visited counterexamples will be harder to find.

In order to solve this problem a state from the current hypothesis is randomly selected from a uniform distribution. The access sequence of the selected state is then used as prefix for the test query. The rest of the test

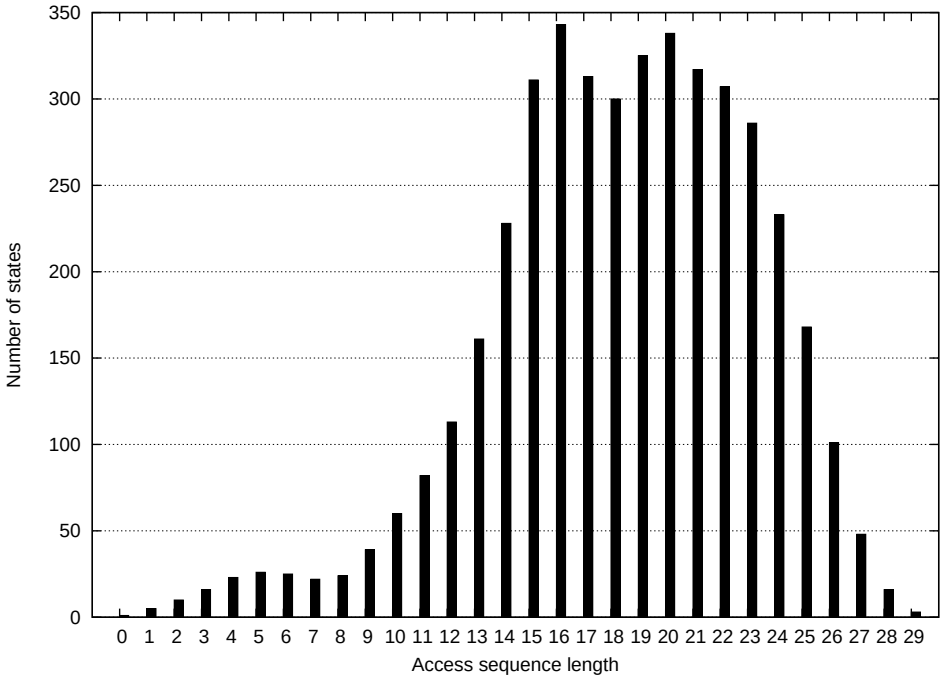

Figure 6: Number of states per access sequence length

query is selected using the random walk algorithm.

The intuition for this technique is that all the states have a roughly equal chance of being visited by a test query. In order to show this more rigorously one could for example use MRMC [KZH+11]. This is however beyond the scope of this thesis.

4.2.2 Subalphabet selection

The random prefix selection technique was used at the beginning of this research. During attempts at learning the ESM it was observed that using this technique some counterexamples were still too hard to find. For example approximately 36 million test queries were needed to find one counterexample and it took more than 2 days to find it.

This prompted a search for a better way to generate test queries. This begins by analysing the reason why the large number of test queries was needed. It was observed that when a counterexample is hard to find the suffix of this counterexample is large. Using random prefix selection there

is a higher chance to find the correct prefix. However for the suffix there is still a large search space. The upper bound for the size of this search space is $|\Sigma|^n$ where $|\Sigma|$ is the size of the alphabet used and n the size of the suffix that needs to be found.

In order find counterexamples with fewer test queries a possible solution is to lower $|\Sigma|$. This was accomplished by dividing the input alphabet Σ in to n subalphabets $\Sigma_1 \dots \Sigma_n$. The union of all these alphabets is again Σ.

The intuition is that for many systems it is possible to reach a large subset of the states using only a small subset of inputs. For example a remote control might have a set of buttons to control the TV and another set of buttons to control the DVD player.

The subalphabets $\Sigma_1 \dots \Sigma_n$ are chosen before the learning begins. This can be done based on knowledge about the SUT, by an algorithm or some combination of this. In order to use these subalphabets the testing phase must be changed. The testing phase is now divided into n parts, one subphase for each subalphabet. In each of these subphases test queries are generated as in random walk but the inputs are selected from subalphabet Σ_n instead of Σ.

In order to decide whether to move from one subphase to the next subphase some heuristic needs to be used. The heuristic that was used during this research is: If $q_{current} > max(q_1, \dots, q_m) * 10$ and $q_{current} > q_{lowerbound}$ then the next subphase is started. Where $q_{current}$ is the current number of test queries executed after the last counterexample was found, $q_1 \dots q_m$ are the number of queries used to find the previous counterexamples in this subphase, and $q_{lowerbound}$ is some fixed minimum lower bound on the number of test queries that needs to be executed. The factor 10 used in this heuristic is based on observations done during the research.

The biggest problem using this technique is finding an effective set of subalphabets. Finding an algorithm that will find an effective set of subalphabets is non trivial. Due to time restrictions no such algorithm was created during this research.

In this case study the subalphabets were chosen by hand. The alphabet Σ was divided into 14 disjoint fragments. These were then composed in order to create 43 subalphabets. The fragments were chosen based on knowledge about the ESM. Each of these 14 fragments represents a functionality of the ESM. By functionality we mean for example changing to a different status (idle, standby, running, defect, etc). For most of these functionalities other functionalities are needed. For example to move to running status the idle

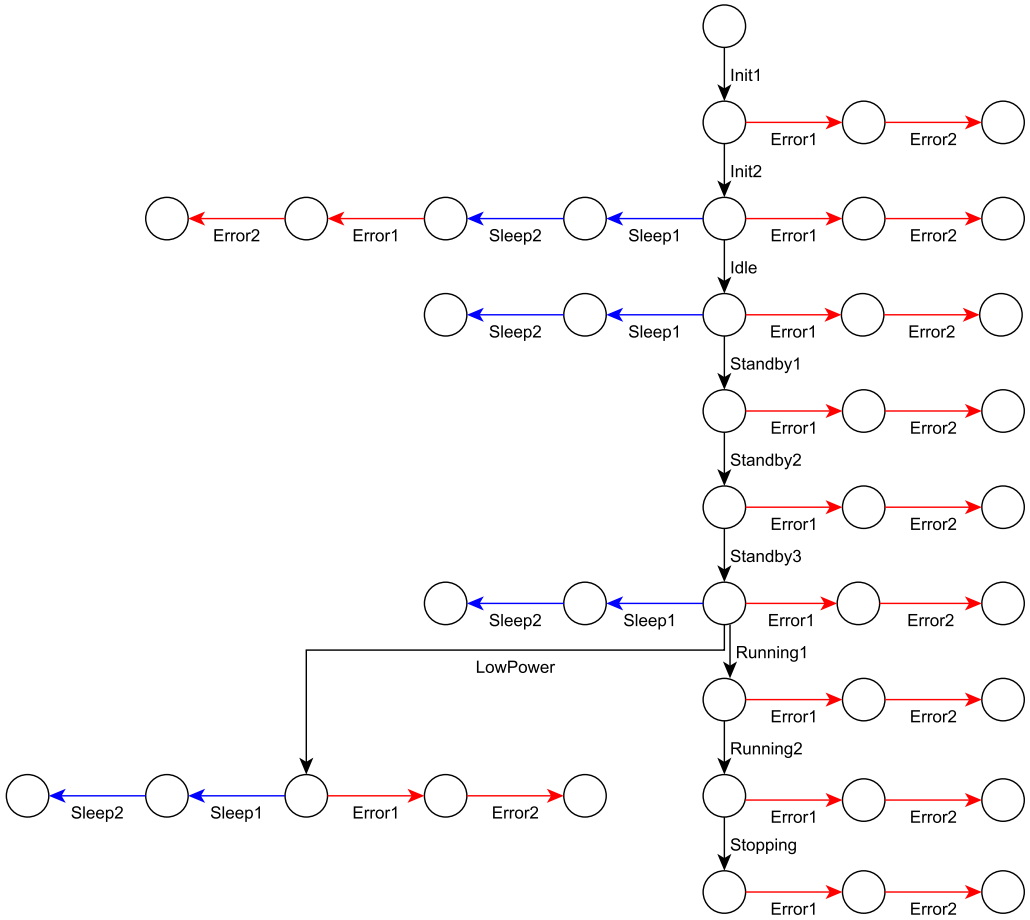

Figure 7: Subalphabets used when learning the ESM

and standby status have to be reached first.

The subalphabets used during the learning of the ESM are shown in figure 7. The vertexes of the tree represent the subalphabets. The root of the tree represents the empty subalphabet. This subalphabet is obviously not used. Each directed edge between two vertexes means that the subalphabet at the target vertex is the subalphabet at the source vertex with the indicated fragment added to it.

4.2.3 Random suffix selection

Another solution for the problems mentioned in the previous section was found. During the learning of the ESM it was observed that the suffix of a counterexample often ends with a suffix found with a previous counterexample.

This observation can be used to find counterexamples using less test queries. When testing begins the set of distinguishing suffixes is extracted from the current hypothesis. This set is maintained during the learning of the model as described in section 3.2. The prefix of the test query is generated using random walk test selection that is build into LearnLib, but a randomly selected suffix from the set of distinguishing suffixes is appended to it.

If this strategy is combined with the random prefix strategy it is basically a variant of the well-known W-method by Chow [Cho78] and Vasilevskii[Vas73]. The test queries are of the form puw, where p is the prefix, w the suffix and u a random sequence of inputs. The difference with the W-method is that p u and w are chosen randomly whereas the W-method generates a list of possible values for p u and w. However this list would be infeasibly large when testing the ESM.

4.2.4 Evolving hypothesis selection

The evolving hypothesis technique described by Howar et al. [HSM10] was used during this research. It is based on the idea that not only the current hypothesis should be used but also all the previous hypotheses.

The proposed algorithm allows LearnLib to refine the hypothesis after each counterexample is found. This means that states and transitions keep the same identity from one hypothesis to the next. During the process of refining the hypothesis states and transitions will be added and deleted.

Using this evolving hypothesis a new test selection strategy can be created. In their paper Howar et al. suggest a blocking strategy. This means randomly choosing a transition from the hypothesis. Then a random suffix is generated. The concatenation of the selected prefix and suffix forms the test query. The selected transition is then blocked from further selection. If all transitions are blocked the algorithm resets by unblocking all the transitions again. The random suffix is generated using the random walk algorithm. Note that the because the hypothesis is refined instead of reconstructed every time a counterexample is found the transitions that are

blocked stay blocked. The result is that new transitions are selected more frequently then existing transitions.

In their paper Howar et al. also suggest a weighted strategy for selecting the transition. A weight for each transition is increased every time the transition is selected. A transition is selected with a probability inversely proportional to the weight. This strategy was however not used because it is more complicated to implement and takes a lot of time each time a test query is generated. During the ZULU competition the blocking and weighted strategies are also shown to need a similar number of test queries.

4.3 Experiments

Experiments were done in order to find out to which extent the strategies discussed above reduce the number test queries needed during learning.

The same set-up as was used for learning the ESM was used to run these experiments. However the process of learning the entire ESM takes too long for experiments to be run with various different test selection strategies. Instead only a partial model of the ESM was learned. This partial model is learned by reducing the input alphabet.

Choosing a suitable partial model of the ESM to learn in these experiments turned out to be difficult. Because all the discussed test selection strategies use randomness the variance in the test results is high. In order to produce meaningful statistics a lot of samples are needed. This poses a restriction on the time used for each run of the experiment. If the chosen partial model of the ESM is too large some counterexamples can not be found within these time constraints. If the chosen partial model is too small some of the selection strategies can not fully show their usefulness.

The restricted input alphabet was that chosen contains the inputs needed for the initialisation and error behaviour. In total this restricted alphabet contains 16 inputs and the model learned from this has 34 states. When using the subalphabet selection strategy the relevant subalphabets are used. They are the same as the subalphabets used when learning the ESM. Each of these subalphabets is a superset of the previous one each adding 5 or 3 inputs.

Even with this restricted alphabet it is hard to learn the model using only random walk in a reasonable time. Because of this the prefix selection strategy is used in each of the experiment set-ups.

The following experiment set-ups were used:

Prefix Only prefix selection strategy.
Suffix Prefix and suffix selection strategies.
Alpha Prefix and subalphabet selection strategies.
EH Prefix and evolving hypothesis selection strategies.
All All of the selection strategies.

For each of these set-ups the model was learned 250 times taking on average about 1,5 minutes per run. Afterwards for each of these runs it was verified that the correct model was learned.

	MQ	TQ no CE found		TQ CE found	
Prefix	14.384	255.748	100,0%	24.001	100,0%
Suffix	14.316	233.566	91,3%	21.537	89,7%
Alpha	14.180	400.000	156,4%	116	0,5%
EH	14.309	231.314	90,4%	20.943	87,3%
All	14.143	400.000	156,4%	177	0,7%

Table 4: Experiment data using Rivest splitter. MQ stands for member queries. TQ stands for test queries. CE stands for counterexample

Table 4 shows the data of these experiments. In the second column shows the average number of member queries. The third and fourth column shows the average number of test queries that did not find a counterexample and the percentage compared to the prefix set-up. The fifth and sixth column shows the average number of test queries that did find a counterexample and the percentage compared to the prefix set-up.

The heuristic discussed in section 4.2.2 is used in these experiments to decide when the learning should be stopped. When using the subalphabet selection strategy this heuristic is used when learning using each of the 4 subalphabets. This heuristic leads to the division in test queries that do an do not find a counterexample.

From this data it can be concluded that when using suffix, subalphabet and evolving hypothesis selection strategy the number of test queries needed to find counterexamples is reduced significantly compared to when only using prefix selection strategy. In case of subalphabet selection strategy a reduction by 2 orders of magnitude can be seen.

However when looking at the test queries needed that do not find a counterexample the subalphabet selection strategy performs worse. These numbers depend on the minimal number of test queries needed after the last

counterexample is found when testing with each of the 4 subalphabets. In these experiments this number was 100.000. This is the same number as used when learning the whole ESM.

When all the strategies are combined the number of test queries needed is not significantly different from the number of test queries needed when using the subalphabet and prefix strategies. This is because of the necessary simplicity of the model that was learned. After the subalphabet strategy is used the other strategies do not add significant speed up when learning the simplified model.

	MQ	TQ no CE found		TQ CE found	
Prefix	9.949	417.559	100,0%	64.910	100,0%
Suffix	9.961	245.950	58,9%	28.882	44,5%
Alpha	9.781	400.000	95,8%	434	0,7%
EH	9.937	355.120	85,0%	55.309	85,2%
All	9,785	400,000	95,8%	506	0,8%

Table 5: Experiment data using Kearns splitter. MQ stands for member queries. TQ stands for test queries. CE stands for counterexample

These 2 experiments were also ran using the Kearns splitter instead of the standard Rivest splitter. This data is shown in table 5. Instead of adding the suffix that is found globally the Kearns splitter only adds the suffix to the table of the corresponding state. This leads to a reduction of the number of member queries needed by 69,3%.

The data also shows that the number of test queries needed is increased when using a Kearns splitter. This is because some counterexamples that are found may be applicable to multiple states. Using a Rivest splitter the counterexample is automatically learned in all states. Using a Kearns splitter the counterexample is only learned for one state. More test queries then have to be spend to find a similar counterexample for other states.

Using the suffix selection strategy this seems to be less of a problem. Only 34,1% more test queries are needed as opposed to 170,4% more when using prefix selection strategy. The reason for this is that once a counterexample is found for one state the counterexamples for the other states are easier to find because the chance of selecting the correct distinguishing suffix is much higher then when using only prefix selection strategy.

4.4 Results

Using the learning set-up discussed in section 4.1 and all the test selection strategies discussed in section 4.2 a model of the ESM using 1 function with 4252 states could be learned. However due to time constraints this model is not correct. During the verification of this model some additional counter-examples were found.

Figure 8 shows the learned model. As can be seen in the figure the number of states and transitions between the states makes it hard to visualise the model. The visualisation was made using Gephi [BHJ09].

In this visualisation the states are coloured according to the strongly connected components they belong to. The edges are coloured based on the inputs and outputs they represent. If multiple transitions from one state to a other state exist this is represented by the thickness of the edge.

The three arms at the top of the figure are three deadlocks in the model. These deadlocks are present in the ESM by design. When the ESM is in such a state it will remain there until the main power supply is turned off. The cluster of states in the top of the figure is the initialisation of the ESM. This is done only once during the execution of the ESM and thus only transitions from this cluster to the main body of states are present.

The following list gives the most important statistics gathered during the learning:

- The learned model has 4252 states.
- The total time needed for learning was 287 hours, 18 minutes and 42 seconds.
- 64,4% of the time was spent executing test queries. The rest was spent executing member queries.
- The total number of member queries was 84.219.652
- The total number of test queries was 131.435.188

During the research and when analysing the experiments it became clear that the number of test queries needed to find a specific counterexample has a lot of variance. However when using a heuristic to decided when to stop executing test queries it is important that the values on which the limit is based don't have a lot of variance.

For example consider a situation where a counterexample would on average need 1.000 test queries to find and it is found after 2.000 test queries. This would mean another 20.000 extra test queries would be needed when on average 10.000 test queries would have been enough.

Figure 8: Learned model of the ESM

5 Verification

In order to verify that the correct model was learnt using LearnLib a model was created by hand.

The ESM is created using RRRT, as described in section 2, and is thus already in a state chart format. This format is however not suitable to prove it equivalent to the learned model. During the research a suitable format to encode the ESM in was searched for.

Any suitable format or tool for this research would need to be able to transform the hand made encoding of the ESM to a label transition system (LTS) or something similar. This can then be compared to the learned result to conclude whether they are equivalent or not.

Some possible candidates for a usable formalism have been found. The first one was hierarchical timed automata (HTA) [DMY02]. Although a tool is available that uses the formalism described in the paper it is hard to translate the output of this tool to something that could be compared to the learned model. Another formalism developed by Hansen et al. [HKL+10] was also considered. Although a tool is also available for use it missed some essential features, for example the ability to assign state variables on transitions. This tool is still under development but could not be used during this research. A formalism called object-oriented action systems (OOAS) [KSA09] has also been described in the literature, but no tools to use this system could be found.

Finally it was decided to create a model and interpret it using our own semantics. The model was created in the UML drawing tool PapyrusUML [LTE+09].

5.1 Model semantics and transformations

In order to model the ESM a model semantic needed to be defined. This semantic needed to be efficient. This efficiency is measured in the number of states and transitions used to define the ESM.

During previous research by the ITALIA project [ITA] the Uppaal [BDL+06] GUI was used as an editor for extended finite state machines (EFSM). Uppaal is a well known model checker that can be used to create timed-automata [AD94] and to check logical properties of these automata.

An EFSM could not be used directly to model the ESM because it is not efficient. The reason for this is that the ESM makes heavy use of

UML [Obj04] state machine concepts such as state hierarchy and transitions from composite states. This leads to a duplication of many transitions and states when modelled using a EFSM.

Instead a hierarchical EFSM (HEFSM) was used. The semantics of a HEFSM have been designed to match the semantics used by the ESM. These semantics are a subset of the semantics used in UML state machines on which the execution model of RRRT is based. Many elements used in UML state machines are left out because they are not needed for modelling the ESM and complicate the transformation process.

The model is transformed to the Uppaal format used in the ITALIA project. Using existing tools within the ITALIA project this transformed model is then transformed again in to a Lotos model. This Lotos model is then compared to the learned model of the ESM using bisimulation available in CADP [GLMS11].

This section will explain the semantics of the HEFSM model created. There will be no formal description of these semantics, this is outside the scope of this research. Instead of the transformation used to transform the HEFSM model to an EFSM model will be explained using examples. The transformation is divided in the following steps and are executed in this order:

- Combine transitions without input or output signal.
- Transform supertransitions.
- Transform internal transitions
- Add input signals that do not generate an output signal.
- Replace invocations of the next function.

In order to make the model more readable and to make it easy to model `if` and `switch` statements in the C++ code the HEFSM model allows for transitions without a signal. These transitions are called *empty* transitions. An empty transition can still contain a guard and an assignment. However these kinds of transitions are only allowed on states that only contain empty outgoing transitions. This was done to make the transformation easy and the model easy to read.

In order to transform a state with empty transitions all the incoming and outgoing transitions are collected. For each combination of incoming transition a and outgoing transition b a new transition c is created with the source of a as source and the target of b as target. The guard for transition c evaluates to true if and only if the guard of a and b both evaluate to true.

The assignment of c is the concatenation of the assignment of a and b. The signal of c will be the signal of a because b cannot have a signal. Once all the new transitions are created all the states with empty transitions are removed together with all their incoming and outgoing transitions.

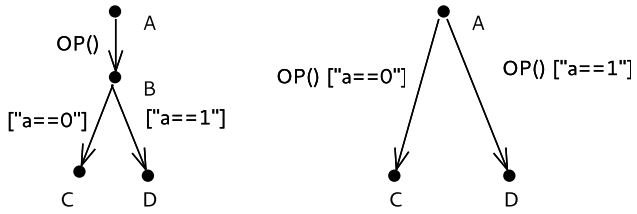

Figure 9: Example of empty transition transformation. On the left the original version. On the right the transformed version

Figure 9 shows an example model with empty transitions and its transformed version. Each of the outgoing transitions from the state `B` are combined with each of the incoming transitions. This results into two new transitions. The old transitions and state `B` are removed.

The RRRT model of the ESM contains many transitions originating from a composite state. Informally this transition can be taken in in each of the substates of the composite state if the guard evaluates to true. These transitions are called *supertransitions*. In order to model the ESM efficiently supertransitions are also supported in the HEFSM model.

In RRRT transitions are evaluated from bottom to top. This means that first the transitions from the leaf state are considered then transitions from it's parent state and then from it's parent's parent state, etc. Once a transition for which the guard evaluates to true and the correct signal has been found it is taken.

In order to transform a supertransition all the states are processed beginning at the top state and recursively going down to its child states. The supertransitions that can be taken in the parent states of a state are collected. If the current state contains a transition a that has the same signal as a supertransition b the guard of b has to be modified because supertransition b may only be taken if transition a cannot be taken. Thus the negation of the guard of a is added to the guard of b.

Once a leaf state is reached all the collected supertransitions are added as outgoing transitions of the leaf state.

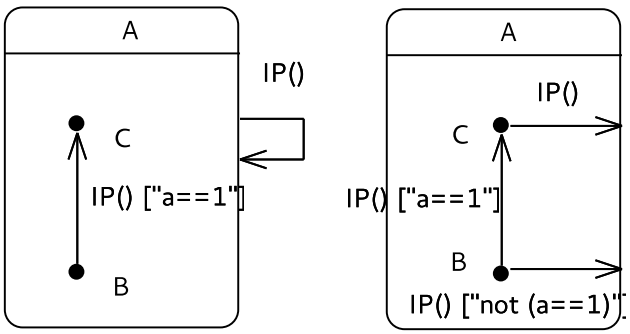

Figure 10: Example of supertransition transformation. On the left the original version. On the right the transformed version

Figure 10 shows an example model with supertransitions and its transformed version. The supertransition from state `A` can be taken at each of `A`'s leaf states `B` and `C`. The transformation removes the original supertransition and creates a new transition at states `B` and `C` using the same target state. For leaf state `C` this is easy because it does not contain a transition with the input signal `IP`. In state `B` the transition to state `C` would be taken if a signal `IP` was processed and the state variable `a` equals 1. The super transition can only be taken if the other transition cannot be taken. This is why the negation of other the guard is added to the new transition. If the original supertransition is an internal transition the model needs further transformation after this transformation. This is described in the next paragraph. If the original supertransition is not an internal transition the new transitions will have the initial state of `A` as target.

The ESM model also makes use of *internal transitions* in RRRT. Using such a transition the current state does not change. If such a transition is defined on a composite state it can be taken from all of the substates and return to the same leaf state it originated from. If defined on a composite state it is thus also a supertransition.

This is also possible in the HEFSM model. In order to transform an internal transition it is first seen as a supertransition and the above transformation is applied. Then the target of the transition is simply set to the leaf state it originates from.

An example can be seen in figure 10. If the supertransition from state `A` is also defined to be an internal transition the transformed version on the right would need another transformation. The new transitions that now have the

target state `A` would be transformed to have the same target state as their current source state.

In order to reduce the number of transitions in the HEFSM model quiescent transitions are added automatically. For every state all the transitions for each signal are collected in a set T. A new self transition a is added for each signal. The guard for transition a evaluates to true if and only if none of the guards of the transactions in T evaluates to true. This makes the HEFSM input complete without having to specify all the transitions.

In RRRT it is possible to write the guard and assignment in C++ code. It is thus possible that the value of a variable changes while an input signal is processed. In the HEFSM however all the assignments only take effect after the input signal is processed. In order simulate this behaviour the *next* function is used. This function takes a variable name and evaluates to the value of this variable after the transition.

5.2 Results

Using the techniques described in the previous section a model was created by hand. This model was compared to the learned model of the ESM. The counterexamples that were found by CADP were given to LearnLib in order to correct the learned model.

Although the entire ESM was modelled some parts may not be entirely correct. The `medium` state and a small part of the rest of the model could not be verified due to time constraints and technical difficulties using CADP. The technical difficulties could not be resolved in time. However the model that was constructed can still be used in other research to improve the learning time of the ESM.

In order to give an indication of the remaining work the manually created model and the learned model were compared. For each the `medium state` was removed and the hand crafted model was transformed to a Mealy machine. The learned model has 3.386 states while the hand crafted model has 4.048. This means that 83,6% of the states were learned.

During the construction of the model the code of the RRRT model was thoroughly inspected. This resulted in the discovery of missing behaviour in one transition of the ESM. An Océ software engineer confirmed that this behaviour was not as expected and has to be analysed further.

6 Conclusion and future work

This research aimed to find out if automata learning is useful in an industrial setting, specifically if it is useful for Océ. In order to investigate this a case study was done. The behaviour of the ESM was learned using LearnLib.

It was expected that learning a correct model of the ESM would be non-trivial but feasible in the available time. However much more time then expected was required in order to find counterexamples.

One reason for this is the size of the input alphabet of the ESM. The large number of possible inputs makes the ESM hard to learn. To solve this the abstracton mapping technique use by Aarts, Jonsson and Uijen [AJU10] could be used. However while studying the ESM no usable abstractions were found.

Another reason is the complexity of the counterexamples that need to be found. Most of the time required to learn the ESM was spent trying to find counterexamples. When this problem was investigated it was discovered that a number of different techniques reduced the number of test queries needed. Some of these techniques such as the subalphabet selection strategy and suffix selection strategies are novel and do not appear in other research that we know of. Data was gathered to show that these techniques indeed reduce the number of test queries needed to find counterexamples. Although these new techniques significantly reduced the testing time it was not enough to learn all the behaviour of the ESM within the available time.

It would be interesting to see how the test techniques used in this research perform when applied to other case studies. The subalphabet selection strategies described in this thesis could be improved by developing an algorithm to select the subalphabets.

In order to show that the correct model was learned a model of the ESM was created by hand. A tool was created in order to transform the model into a Lotos model. This Lotos model could then be compared with the learned model. Although a large part of the learned model of the ESM has been proven correct the work could not be finished in time.

There are several interesting possibilities for future research. The ESM has proven to be an interesting research subject. No correct model of the ESM using one function has been learned yet. In order to reach this goal the time needed to learn the ESM needs to be reduced. Using an obfuscated version of the model created by hand this research can be continued outside of Océ. The model can also be used as a benchmark for other research groups

to compare different learning and model-based testing techniques.

There are some other opportunities for research within Océ. A model of the ESM using more than 1 function could also be learned. Another interesting possibility is to learn models of the EHM, ACM and other managers connected to the ESM. Using these models some of the properties discussed by Ploeger [Plo05] could be verified at a more detailed level.

We conclude that LearnLib cannot learn a correct model of the ESM. This is not a problem related to the learning algorithms implemented in LearnLib but rather the inability to find counterexamples using model-based testing algorithms. The main contribution of this research are the novel test selection techniques which greatly improve the ability to find counterexamples for the incorrect hypothesis of the ESM. The case study described in this thesis can serve as a benchmark for the automata learning and model-based testing community.

References

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[AJU10] Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In *ICTSS*, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2010.

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, November 1987.

[AV10] Fides Aarts and Frits W. Vaandrager. Learning I/O automata. In *CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2010.

[BDL+06] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. Uppaal 4.0. In *QEST*, pages 125–126. IEEE Computer Society, 2006.

[Bel10] Axel Belinfante. JTorX: A tool for on-line model-driven test derivation and execution. In *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer, 2010.

[BHJ09] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks. In *ICWSM*. The AAAI Press, 2009.

[Boh09] Therese Bohlin. *Regular inference for communication protocol entities*. PhD thesis, Uppsala University, 2009.

[Cho78] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.

[DMY02] Alexandre David, M. Möller, and Wang Yi. Formal verification of UML statecharts with real-time extensions. pages 208–241, 2002.

[GLMS11] Hubert Garavel, Frdric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2010: A toolbox for the construction and analysis of distributed processes. In *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer, 2011.

[GvD07] Bas Graaf and Arie van Deursen. Model-driven consistency checking of behavioural specifications. In *MOMPES*, pages 115–126. IEEE Computer Society, 2007.

[HKL+10] Helle Hvid Hansen, Jeroen Ketema, Bas Luttik, Mohammad Reza Mousavi, Jaco van de Pol, and Osmar Marchi dos Santos. Automated verification of executable UML models. In *FMCO*, volume 6957 of *Lecture Notes in Computer Science*, pages 225–250. Springer, 2010.

[HSM10] Falk Howar, Bernhard Steffen, and Maik Merten. From ZULU to rers - lessons learned in the zulu challenge. In *ISoLA*, volume 6415 of *Lecture Notes in Computer Science*, pages 687–704. Springer, 2010.

[ITA] ITALIA project. `http://www.italia.cs.ru.nl/`. Accessed: 23/07/2012.

[KSA09] Willibald Krenn, Rupert Schlick, and Bernhard K. Aichernig. Mapping UML to labeled transition systems for test-case generation - a translation via object-oriented action systems. In *FMCO*, volume 6286 of *Lecture Notes in Computer Science*, pages 186–207. Springer, 2009.

[KZH+11] Joost-Pieter Katoen, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N. Jansen. The ins and outs of the probabilistic model checker MRMC. *Performance evaluation*, 68(2):90–104, 2011.

[LTE+09] A. Lanusse, Y. Tanguy, H. Espinoza, C. Mraidha, S. Gerard, P. Tessier, R. Schnekenburger, H. Dubois, and F. Terrier. Papyrus UML: an open source toolset for MDA. In *Fifth European Conference on Model-Driven Architecture Foundations and Applications*, page 1, 2009.

[MSHM11] Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next generation LearnLib. In *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 220–223. Springer, 2011.

[Nie03] Oliver Niese. *An integrated approach to testing complex systems.* PhD thesis, Dortmund University of Technology, 2003.

[Obj04] Object Management Group (OMG). Unified modeling language specification: Version 2, revised final adopted specification. `http://www.uml.org/#UML2.0`, 2004. Accessed: 23/07/2012.

[Plo05] Bas Ploeger. Analysis of concurrent state machines in embedded copier software. Master's thesis, Eindhoven University of Technology, August 2005.

[RSBM09] Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. LearnLib: a framework for extrapolating behavioral models. *STTT*, 11(5):393–407, 2009.

[SHM11] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In *SFM*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer, 2011.

[Tre08] Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[Vas73] M. P. Vasilevskii. Failure diagnosis of automata. *Cybernetics and Systems Analysis*, 9(4):653–665, 1973.