Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Learning Fresh-Register Automata

Yasir Zardari
Supervisor: Dr. Vasileios Koutavas

April 2020

A Dissertation submitted in partial fulfilment
of the requirements for the degree of
Master in Computer Science

Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

Signed: Yasir Zardari Date: 30-04-2020

Abstract

This dissertation investigates if fresh-register automata can be learned via existing automata learning techniques. Fresh-register automata are a relatively new class of automata, capable of encoding the properties of π-calculus models, which are used to describe the essential behaviours of concurrent systems.

This was attempted by creating a system which maps fresh-register automata to register automata models, before learning with the register automata learning tool Tomte and finally mapping back to fresh-register automata.

Results from this approach revealed that fresh-register automata models could not be consistently learned, mainly due to features regarding the learning of register automata by Tomte.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor, Dr. Vasileios Koutavas, for his constant guidance and support throughout the duration of this research. Without his time and patience, this dissertation would not have been possible.

I would also like to express my utmost gratitude to my parents for their endless support throughout my academic journey.

Last but not least, I want to say thank you to the friends I have made over the last five years for their constant support and encouragement.

Contents

List of Figures

List of Tables

1 Introduction

When presented with various software systems and components, multiple methods of obtaining its behaviour and functionality are available. Some of the more common and obvious ones include reading the appropriate documentation or exploring the source code of said software component. However, these resources are not always available to us, making it difficult to determine and analyse a system's behaviour, a relevant issue considering the increasing complexity of today's software.

An approach which has received a lot of study in recent times is the ability to infer models of software systems in a black-box context, namely models in the form of *state diagrams*, which are very effective in describing the abstract behaviour of a system. Algorithms have been designed to carry out this, with the first being introduced by Dana Angluin in the form of her *L** algorithm [2]. This approach is referred to as Automata Learning.

1.1 Motivation

In terms of black-box learning of systems, automata learning is considered dual with the process of model checking [3]. The former's goal is to infer a model through interaction with the system by providing inputs and observing outputs, while the latter implies exploring the state space and building the model by hand, before verifying if the system matches the model. While the employment of model checking has been around much longer, automata learning has seen rapid progression in recent years, with the continued design of newer algorithms to handle the learning of richer classes of models, including those that handle data.

Automata learning has been successfully applied across many industrial areas where it has proven to be an extremely effective verification and bug finding technique. Some notable examples include a study by Aarts et al. on the learning of the EMV banking card models [4], as well as the use of automata learning on the TLS protocol implementations by De Ruiter and Poll [5], which revealed security issues regarding the Java Secure Socket Extension. Another example is a case study by Fiterau et al. [6], who used automata learning and model checking to demonstrate that Linux, Windows, and FreeBSD

implementations of TCP did not conform to their RFC specifications.

Despite the remarkable advancement of automata learning in recent decades, there still exists classes of automata for which learning algorithms have not yet been designed. One examples is the relatively new fresh-register automata (FRA) [7]. FRAs have been shown to encode the behaviour of π-calculus processes, which are concurrent communicating processes whose behaviours are modelled by an established calculi known as π-calculus [8]. π-calculus is typically geared towards model checking. However, due to emergence of fresh-register automata which can capture the properties of π-calculus models, it is feasible to suggest that π-calculus systems can be learned via the employment of automata learning on FRA models.

1.2 Dissertation Objectives

The objective of this dissertation is to explore the possibility of learning π-calculus models via the learning of fresh-register automata. This will be done by adapting existing automata learning techniques, specifically register automata learning, a class of automata very closely related to fresh-register automata. This will follow a study on a basic regular set (deterministic finite automata) learning algorithm [2]. A more detailed breakdown of the objectives is as follows:

1. **Understand the basics of automata learning**

   The first goal of this dissertation is to explore the first automata learning algorithm, named L*, devised by Dana Angluin in her paper in 1987 [2]. This would involve implementing the algorithm programmatically, in doing so achieving a greater understanding of automata learning concepts considering the influence this work has had in future algorithms within the field.

   Performance testing will then be carried out on the L* implementation, identifying possible bottlenecks associated with Angluin's original implementation, as well as exploring possible improvements to ratify these performance issues.

2. **Understand register automata learning algorithms**

   The next objective will be to understand in greater detail existing register automata learning techniques. Analysis will be done on an existing register automata learning tool named Tomte [9], which will be used in an attempt to apply learning to fresh-register automata. This will involve a study on how the tool works in both a theoretical sense and practical sense.

3. **Develop fresh-register automata learning technique**

   The third goal of the dissertation is to study and understand a class of automata known as FRA, or fresh-register automata [7]. This type of automata is closely related to register automata (RA), mentioned in the previous objective.

   Due to this similarity, the idea would be to adapt and apply a learning algorithm for register automata to accommodate FRA's, for which there does not currently exist automata learning algorithms. This would involve creating tools to map FRA models we wish to learn into a register automata equivalent, before learning these automata using the tool Tomte (mentioned above) and finally mapping back to an FRA, with the hope that this model is equivalent to the original FRA.

1.3 Dissertation Structure

- Chapter 1 details the motivation for the dissertation and a list of goals/objectives, as well as a possible solution to the research problem of learning fresh-register automata.
- Chapter 2 describes the relevant work done in the field, which includes work done on earlier automata and their learning algorithms as well a brief look at fresh-register automata and π-calculus in general.
- Chapter 3 explores basic automata learning, specifically looking at the L* algorithm, a DFA learning algorithm. An implementation of this algorithm is described as well as performance testing done for this research project.
- Chapter 4 looks at the workings of the register automata learning tool Tomte, due to its important role in the FRA learning architecture developed in the subsequent chapters.
- Chapters 5 and 6 outline the FRA learning process developed in the aim of solving the already described research problem. Two attempts are described, which includes details on the architecture, implementation and evaluation of the system.
- Chapter 7 gives some concluding thoughts on the research project as a whole, describing the results in addition to possible future work.

2 Related Work

As mentioned in the previous chapter, automata learning has been a growing research topic for the last number of decades, with a number of learning algorithms and techniques being designed in recent years in parallel to the emergence of more complex and sophisticated software systems. This chapter will look at publications which have contributed significantly to the field in terms of learning algorithms, from those dealing with simple DFA's, to richer classes involving data (register automata).

2.1 Basics of Automata Learning: L* Algorithm

In 1987, Dana Angluin published her paper titled "Learning Regular Sets from Queries and Counterexamples" [2]. In this, she was able to prove that (unknown) regular languages or deterministic finite automata (DFA) could be inferred, in what could be seen as the first use of active automata learning. This is done via the use of queries, which the *learner*, in this case the L* algorithm, is able to ask the automata being learned, which is appropriately titled as the *teacher*. This forms the basis of the *Minimally Adequate Teacher* or MAT model, which Angluin introduces in her paper and which is used in many automata learning algorithms designed afterwards.

The teacher is able to answer two types of queries from the learner, membership queries and equivalence queries. Membership queries test whether an input is part of the unknown regular language being learned. These queries are adopted to build a hypothesised DFA model. Equivalence queries are then employed to test whether the hypothesised model is equivalent to the model under learning. If so, the algorithm terminates, otherwise a counterexample is returned, which helps to build another hypothesis and so on. A polynomial number of these queries are usually asked during the entire run of the algorithm. A more comprehensive description of the L* algorithm is found in chapter 3.

L* has crucially been used in black-box testing and verification techniques, which as mentioned in the introductory chapter has been an important field for the use of automata

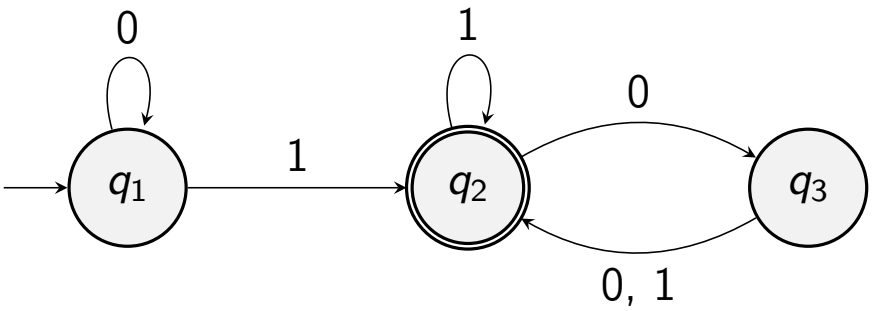

Figure 2.1: Example of a DFA

learning techniques. Peled et al. were able to utilise the MAT framework devised by Angluin to develop techniques that were capable of carrying out extensive *black box checking* and conformance testing [10]. However, Berg et al. notes the lack of reported efforts of Angluin's L* algorithm being used to learn the finite automata of reactive systems, describing how it is still not possible to make effective conclusions over the algorithm's efficiency over such systems [11]. Inevitably, since Angluin's paper's publication, faster variations of the algorithm have been designed as well as newer algorithms to accommodate for the learning of higher complexity systems, with the higher efficiency variants closely following Angluin's MAT framework, highlighting the influences of her research within the field.

2.1.1 Learning Mealy Machines

A well known variation of the original L* algorithm is its adaption to Mealy machines. Mealy machines, though quite similar to the deterministic finite automata model, are much more suited to describing reactive systems due to the fact they are transducers and not acceptors like regular DFA. Examples of possible applications include vending machines and traffic lights due to their multiple concurrent subsystems.

Like a DFA, a Mealy machine contains a set of states, only instead of accepting or rejecting input, a Mealy machine outputs at each step a finite symbol from a finite alphabet, hence defining it as a mapping of input symbols to strings of output symbols. A more formal description of a Mealy machine is as follows [12]:

Definition 1: A **Mealy machine** is a tuple M = $\langle Q, q^0, \sum, \Omega, \delta, \lambda \rangle$ where:

- Q: Nonempty finite set of states
- q^0: The initial state where $q^0 \in Q$
- $\sum$: Nonempty finite input alphabet
- Ω: Nonempty finite output alphabet

- δ: $Q \times \sum \rightarrow Q$
- λ: $Q \times \sum \rightarrow \Omega$

A simple Mealy machine is shown below with the alphabet of {0,1}. The initial state is q^0, where the output is the exclusive OR of the two most recent inputs. For example an input of 0010 would return an output of 0011.

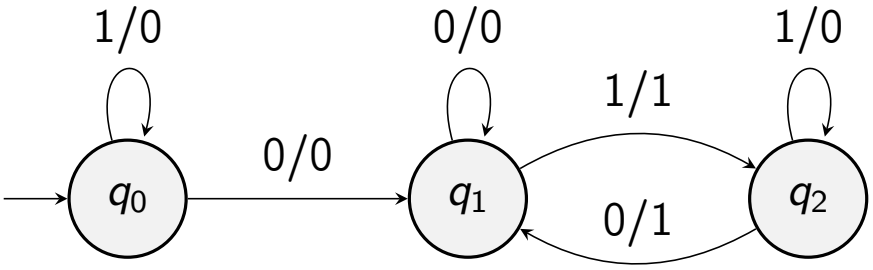

Figure 2.2: XOR Mealy Machine

Adaption of the L* algorithm to learn Mealy machines was first devised by Niese in [13] to allow for learning of application specific systems. This was further analaysed by Shahbaz and Groz [14]. The main idea of this was instead of storing the answers to membership queries as either "yes" or "no", the adapted algorithm would store in a table the outputted word of an input, akin to the difference of a DFA and Mealy machine as described above. The addition of storing more complex information can lead to negative performances in the case of systems of large alphabets which are common in real-life systems, however, improvements to the algorithm have been made in subsequent years in future publications [15].

2.2 Learning Register Automata

While many advances have been made to the standard L* algorithm, they're not suitable to handle learning of much more complex and sophisticated industry standard software and hardware systems. Such components rely on richer classes of automata for modelling their behaviour, those that allow for data values to be stored and manipulated. One such class are the relatively new register automata [16]. Register automata are an extension to the standard DFA, being equipped with a finite set of registers, in which data values that range over infinite domains (integers, strings etc.), can be stored as well as compared for equality, inequality etc. This functionality makes register automata suitable for modelling a login

system, where certain actions (parameterised input symbols) such as "create_account(p1,p2)" requires values to be stored in registers, and "login(p1,p2)" to compare supplied parameters with the existing register values. Other such applications include the modelling of data structures such as a stack or a priority queue, which both involve the storing of data values.

Register automata can come in different variations such as the regular DFA or Mealy machine standard, with the exception being the latter allowing for output symbols, as differentiated in section 2.1.1. For the purposes of this dissertation, we will be assuming the Mealy machine version of the register automata, for which a full definition is defined below [1]:

Definition 2: A **register automaton** is a tuple R = $\langle L, l^0, I, X, T \rangle$ where:

- L: Nonempty finite set of locations (states)
- l^0: The initial location where $q^0 \in Q$
- I: Finite set of input symbols
- O: Finite set of output symbols
- X: Nonempty finite set of registers
- T: Finite set of transitions. Each transition is of the form $\langle l, i, g, \sigma, o, l' \rangle$ where l is the source location, i is the paramaterised input, g is the guard, σ is the assignment statement and l' is the target location.

Below is an example of a register automaton modelling a first-in, first-out (FIFO) data structure or a "queue" [1]. Here, values can be pushed onto the queue as well as popping values off the queue in the same order they are added. In this scenario, the set of input symbols would contain a "Push(x)" action, which inserts a value into the queue (store value in a register), hence the parameter "x", and the "Pop()" action, which returns a value. Different output symbols can occur depending on the action and condition (also known as the *guard*). For example, assuming a queue capacity of two, the output of a successful Push would be "OK", otherwise, if the queue is full, then the output would be "NOK" to indicate unsuccessfulness. For Pop, the output would be "Return" with the parameter being the oldest value in the queue, or "NOK" if the queue is empty. Duplicate values cannot be pushed in this example, again returning "NOK".

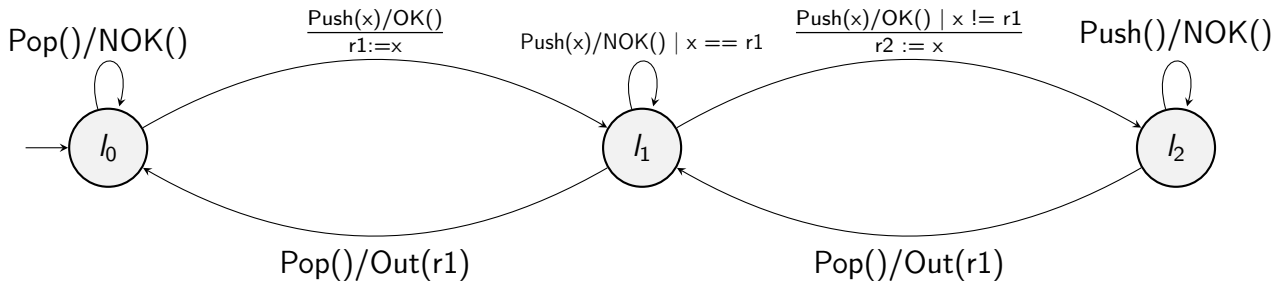

Figure 2.3: FIFO Register Automata

A number of different techniques have been devised to learn these register automata models. One such method was designed by Cassel et al. [17]. In this publication, they present their own active learning algorithm to infer canonical register automata. The algorithm is based on the classic L* algorithm for DFA models, using basic queries of membership and equivalence as well as similarly storing these query results in an observation table. The same pattern is followed in both algorithms, with hypothesised automata being inferred until the algorithm terminates. The key difference between these algorithms is the ability for the newer algorithm to learn the required registers and guards found in transitions. It does this by formulating a Nerode congruence for register automata in order to identify this extra information. The Nerode relation is used in automata learning algorithms in order to identify the states and transitions of a learned automaton from the observation table. The same authors updated their approach in to create the SL* algorithm [18], which makes use of symbolic decision trees instead of membership queries, which summarises the results of tests in a symbolic fashion.

Another approach to learning register automata is explored by Aarts et al. [1]. This algorithm uses a "counterexample guided abstraction refinement" to automatically map the set of actions of the register automata being learned to a set of actions that can be handled by a Mealy machine learner. The algorithm is implemented by the authors in a tool called Tomte (explained in more detail in chapter 4). This particular implementation also allows for the generation of fresh output values, which are essential in systems which generate new data eg. password creators.

2.3 π-calculus

π-calculus belongs to the family of process calculi, which are minimal languages used to formally model the essential behaviours and properties of concurrent systems.

π-calculus was designed by Milner et al. [8] in the aim of rigorously describing, using a small number of basic concepts, the concurrently active components of *interactive* systems. How π-calculus does this is by the generation of *channels*, which can be used as a communication

link between two processes.

These channels are in the form of entities known as *names*. Names can be created at will and in a way that newly created names are distinct from any existing entity, in other words, names are always *fresh*. This can be evident in the creation of a *fresh* communication channel between two processes in a concurrent system. This freshness quality allows for an important feature of π-calculus, which is the creation of restricted or local channels, that cannot be accessed outside the initial declared set of processes, with the name being distinct from all other names in all processes.

No such automata class existed which could model the properties of concurrent system described by the π-calculus. This was until the emergence of the fresh-register automata (FRA) in 2010, which could model the behaviour of these processes in terms of creating locally, receiving or sending *names*. Fresh-register automata were designed to specifically model the basic aspects of *names* and in turn π-calculus systems which relied heavily on them.

2.4 Fresh-Register Automata

Fresh-register automata (FRA) are a new class of automata introduced by Nikos Tzevelokos [7]. FRA models are quite similar to the previously discussed register automata, in the fact they also use a finite number of registers to store incoming data values and compare them with previously stored ones. The key feature of fresh-register automata is its ability to recognise *fresh* inputs, both locally and globally. Locally fresh inputs refer to inputs that are not currently stored in any of the available registers, while global freshness indicates that the automata can accept inputs that are fresh in the whole run i.e. have never been stored in the automaton's registers up to that point.

Definition 1: A **fresh-register automata (FRA)** of n registers is a tuple $A = \langle Q, q^0, \sigma^0, \delta, F \rangle$ where:

- Q: Nonempty finite set of states
- q^0: The initial state where $q^0 \in Q$
- σ^0: Initial register assignment
- δ: Transition relation
- F: Set of final states

The three different types of transitions in an FRA are represented as follows:

- **Local Freshness**: An input is locally fresh if it is not already stored in any of the registers. For example, in an FRA the transition

$$q \xrightarrow{i\,\bullet} q'$$

means that the FRA in question is at state q and if the input name does not exist within the set of registers, then the FRA can proceed to q'.

- **Global Freshness**: An input is globally fresh if is fresh in the whole run i.e. has not previously been stored in any of the registers. This is represented as a transition in an FRA as:

$$q \xrightarrow{i\,\circledast} q'$$

- **Known transition**: an FRA can also contain a *known* transition, where the input is already found within a register. This is represented as follows:

$$q \xrightarrow{i} q'$$

The following is an example of an FRA model from [7] which makes an initial assignment of {1,#} (This can be construed as the initialisation of register 1). This automaton receives a name a and keeps receiving it until a name b where $a \neq b$. It then keeps receiving b until a globally fresh c is received, before it transitions back to the start. *Names* are defined as special programming entities which can be created at will and are always *fresh*.

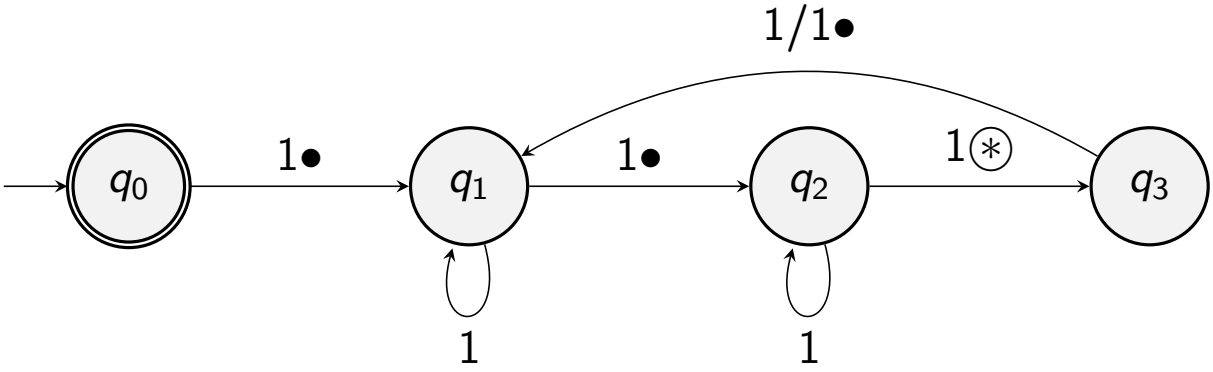

Figure 2.4: Fresh-Register Automaton Example

From the above example, we can see that an FRA model is very similar to a register automaton with regards to the storing and manipulating of values in registers. In fact, a register automata model is the same as an FRA model, except a register automaton does

not recognise globally fresh inputs and as such does not contain globally fresh transitions. Due to this similarity it was decided to make use of register automata learning techniques in an attempt to devise a method of learning fresh-register automata.

Fresh-register automata have been shown to be able to encode the behaviour of π-calculus processes [8]. π-calculus is used in the area of process algebra in order to describe concurrent systems and mobile processes such as network protocols and secure transaction which involve the use of freshly created entities. From this it can be deduced that in order to learn pi calculus processes, one can infer the fresh-register automata models for which they can be encoded into. This would require developing a technique based on current automata learning principals and methods in the aim of learning these type of automata (FRA), which is the main aim of this dissertation (chapter 5/6).

2.5 Other Relevant Work

Apart from the work described so far in this chapter, the field of automata learning has spread to many other classes of automata models. One such example is the learning of nominal automata [19], which are automata which accept languages over infinite alphabets. Like FRAs, nominal automata heavily relate to the concept of names and are inspired by the work of π-calculus. This study [19] generalises the L* algorithm, like many other examples, to develop an algorithm capable of learning nominal automata.

3 Basics of Automata Learning: L* Algorithm

For the purposes of this research, it was important to understand the basics of automata learning and its origins, as many of the same techniques have been used throughout the last few decades in this field of research, as was observed in the previous chapter. This chapter takes a deeper look into one of the original automata learning algorithms, the L* algorithm, which was also implemented and performance tested to obtain a greater understanding of automata learning.

3.1 L* Algorithm Explained

In 1987, Dana Angluin published her paper titled "Learning Regular Sets from Queries and Counterexamples" [2]. In this paper, she introduced the L* algorithm, which is able to learn finite automata through the use of queries, specifically membership queries and equivalence queries. The algorithm assumes an approach involving a *Minimally Adequate Teacher* (MAT), which can answer these two types of queries, asked by the *Learner* in order to learn an unknown regular set. The MAT model, first introduced here, has inspired a number of faster and more efficient variations since the paper's publication.

3.1.1 The Minimally Adequate Teacher

As mentioned before, Angluin introduced the concept of a Minimally Adequate Teacher, which is assumed to correctly answer two types of queries asked by the Learner (which refers to the learning algorithm):

- **Membership Query**: The Learner asks a string t and the Teacher answers whether t is part of the regular set.
- **Equivalence Query**: The Teacher answers whether a hypothesised DFA by the Learner is equal to the unknown regular set. If yes, then the algorithm terminates.

Otherwise, the Teacher responds with a counterexample, which is a string accepted by one DFA but not the other.

In this setting, the learning algorithm L* is able to learn the unknown regular set in a polynomial number of membership and equivalence queries.

3.1.2 The Algorithm

Throughout each step of the algorithm, the Learner keeps information regarding a finite collection of strings over the alphabet A. This is organised into a tabular format, referred to as the *observation table*. The observation table's rows denote a finite, prefix-closed set S, unioned with the one character extension of S, named $S{\cdot}A$

$$\text{Rows} = \{t \mid t \in S \text{ or } t \in S{\cdot}A\}$$

while the columns denote a finite suffix-closed set E. An entry in the table is marked as 1 if the string $s \in$ Rows concatenated with the string $e \in E$ is part of the unknown set. Otherwise it's marked with a 0. This is done with the use of membership queries.

	λ
λ	1
0	1
1	0

Table 3.1: Example of an Observation Table

The observation table equates to a particular deterministic finite automaton, with the rows S representing the state set, the empty string λ row being the initial state, the set of rows in S which belong to the unknown language as the accepting states and finally the transitions as δ(row(s),a) = row(s·a). There are two conditions for which the observation table must meet to allow the construction of a dfa:

- The table must be **closed**: For each row in $S{\cdot}A$, there exists an equal row in S. As each row represents a state in the DFA, closedness allows us to define transitions from each state for each letter in the alphabet A.

	λ
λ	1
1	0
0	1
10	0
11	1

Table 3.2: Example of a Closed Observation Table

- The table must be **consistent**: If two rows for elements *s1* and *s2* within S are equal (identical membership query results across row), then the rows $s1 \cdot a$ and $s2 \cdot a$ for all A must also be equal. This ensures that the transition of the equivalent two rows by a letter lead to the same target state.

	λ
λ	1
0	1
1	0

Table 3.3: Example of a Consistent Observation Table

At the beginning of the algorithm the observation table is initialised with S and E equal to the empty string λ. Membership queries are then asked for λ and each a in A.

The algorithm then enters into the main loop, which repeats until the table is both consistent and closed.

If the current observation table is found to be not consistent, the $a \cdot e$ where row($s1$) = row($s2$) and table entry $s1 \cdot a \cdot e \neq s2 \cdot a \cdot e$, is added to E. If the table is found to be not closed, the $s \cdot a$ where row($s \cdot a$) is different from row(s) for all $s \in S$, is added to S. Any holes in the table created by these alterations are subsequently filled using membership queries.

Once the observation table is closed and consistent, a DFA hypothesis is constructed and an equivalence query is made (how this equivalence query is made is not part of the algorithm, and in practice different approaches exist (section 3.4.2)).

Each row(s) where $s \in S$ denotes a state in the automaton, with the initial state being row(λ). Final states are represented by row(s) where the membership query of s is 1. Angluin denotes the transition function as δ(row(s,a) = row($s \cdot a$)

After an equivalence query is called, if the teacher replies with a counterexample t, then t and its prefixes are added to S, with any holes filled with membership queries. The table is again checked to see if it's consistent and closed. This cycle is repeated until the equivalence query replies with *yes* and the algorithm terminates.

3.1.3 Example Run of L*

For this example we will assume the unknown language is the set of all strings whose binary representation is a multiple of 3, with $A = \{0,1\}$. For example, the string 1100 would be

accepted as it equates to the decimal number 12 (divisible by 3), unlike string 1010, which is equal to 10 (not divisible by 3).

We first start with the initial observation table of $S = E = \lambda$. Membership queries for rows $\lambda \cup A$ are asked by the Learner, giving the following:

	λ
λ	1
0	1
1	0

This table is consistent with only one row $\in S$, but not closed, as row(1) differs to row(λ). As per the algorithm, 1 is added to S and the new extension $S{\cdot}A$ is computed before membership queries are carried out for the new missing elements. The new set of rows are shown in the table below.

	λ
λ	1
1	0
0	1
10	0
11	1

The current observation table is closed and consistent, meaning a DFA can be constructed and thus an equivalence query is made.

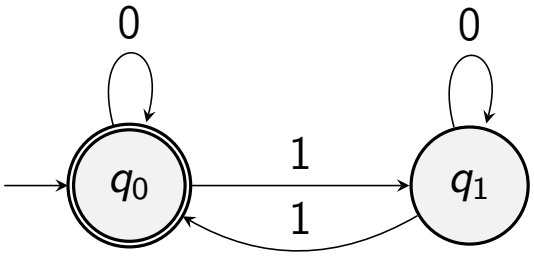

Figure 3.1: First Hypothesis

This hypothesis is rejected by the Teacher, with the counterexample of 101. As per the algorithm description, strings 1, 10 and 101 are added to S, and as before, membership queries are computed to fill the table.

	λ
λ	1
1	0
10	0
101	0
0	1
11	1
100	0
1010	0
1011	0

This observation table is closed but not consistent, as row(1) = row(10), but row(11) $\neq$ row(101). As a result, 1 is added to E and the table is reconstructed once more.

	λ	1
λ	1	0
1	0	1
10	0	0
101	0	0
0	1	0
11	1	0
100	0	1
1010	0	1
1011	0	0

Now that the table is once again closed and consistent, a dfa is constructed and another equivalence query is carried out. This time the Teacher replies yes, marking the end of the algorithm.

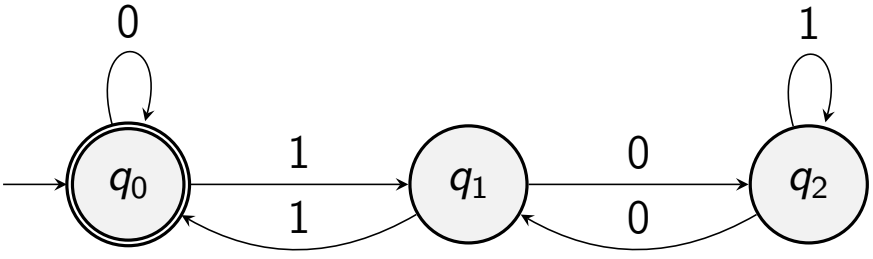

Figure 3.2: Second Hypothesis

3.2 L* Implementation

A simple implementation of the algorithm was written in order to increase my understanding of it. The implementation was in the form of a simple Python program [20]. The programming language Python was chosen due to its ease of use and simple syntax, as well as its large collection of libraries, some of which were used in the actual program eg. automaton library and Graphviz library (graph visualisation).

3.2.1 Design

The implementation attempted to follow the actual algorithm as accurately as possible. Using an object oriented style, two classes were created to represent both entities in the MAT model, the Learner, which would attempt to infer the unknown DFA using queries, and the Teacher, which has knowledge of the DFA being learned and would respond to said queries.

3.2.2 DFA Definition

Using an imported library, a simple DFA object could be easily defined and used for testing. Information entered within the object consisted of names of states, input symbols or alphabet, state transitions (in the form of a Python dictionary), initial state and final state. An example of such a definition is shown below:

```
dfa_to_learn = DFA(
    states={'q0', 'q1', 'q2','q3'},
    input_symbols={'0', '1'},
    transitions={
        'q0': {'0': 'q1', '1': 'q3'},
        'q1': {'0': 'q0', '1': 'q2'},
        'q2': {'0': 'q3', '1': 'q1'},
        'q3': {'0': 'q2', '1': 'q0'},
    },
    initial_state='q0',
    final_states={'q0'}
)
```

Figure 3.3: DFA Object

3.2.3 Learner Class

The main purpose of the learner class, as mentioned above, would be to attempt to infer the unknown DFA or regular set/language, through the use of membership queries and equivalence queries. As the class acted as the role of the 'learner' in the MAT model, it would essentially follow the algorithm described in Angluin's paper, step-by-step.

As its role as the learner, the class would also have to maintain the observation table, pinnacle to the running of the algorithm. In order to simplify the implementation, data structures in the forms of a list were used to represent the different parts of the observation table, namely a separate list for the table's rows (set S and set $S{\cdot}A$) and columns (set E). Entries of the table were represented by the dictionary data structure, with the index for each dictionary entry being the concatenation of the row and column label, matching the structure of the observation table described in section 3.2.2.

The main algorithm was implemented in a class method named "learn". Here, the observation table is initialised, with membership queries (part of the Teacher class) being asked for the empty string (λ) and each input symbol. The main loop is a simple while loop, which continuously runs until the table is closed and consistent. Shown below is an example of the consistent check code and how the observation table is accessed. The two opening for loops iterate through every combination of 2 rows in the table. For each of these two, it is first checked if the rows are equal, as stated in the algorithm. If equal, then each of the rows appended with a symbol from the DFA alphabet (via for loop) is checked. If a difference is found, then the function returns false for not consistent. If every possibility has been exhausted, the function returns true for the table being consistent. The closedness check code is written in a similar manner in terms of for-loops for row access.

Separate methods were implemented to check for these conditions, returning booleans of either true or false. After the loop, a method then converts the observation table lists information into a DFA object, which is then passed to the Teacher object's equivalence query method, returning either the string "equivalent", after which using the Graphviz library a visualisation of the learned DFA is presented to the user and the program terminates. Otherwise, a counterexample is returned which is added to the observation table and the program returns to the initial while loop.

3.2.4 Teacher Class

The teacher class would have access to the unknown DFA and have the ability to answer queries asked by the learner. As such, two methods are implemented within this class, an membership query function and an equivalence query function. The membership query consists of a simple if statement, checking whether the query string is part of the unknown

```
def isConsistent(self):

    for s1 in S:
        for s2 in S:
            rowS1 = [T[s1+e] for e in E]
            rowS2 = [T[s2+e] for e in E]

            if rowS1 == rowS2:
                for a in alphabet:
                    newRowS1 = [T[s1+a+e] for e in E]
                    newRowS2 = [T[s2+a+e] for e in E]

                    if newRowS1 != newRowS2:
                        return False

    return True
```

Figure 3.4: Consistency Check Code

DFA or not. Minimal code is required here as the import automata library provides a function to check for acceptance.

In the case of the equivalence query, a brute force method is employed to check whether the hypothesis and unknown DFA are equivalent. It does this by checking whether every permutation of the DFA alphabet up to a certain length returns the same result (accept or reject) for both the hypothesis and unknown DFA. The length mentioned here was chosen to be the number of states in the unknown DFA. If every permutation has been exhausted and no conflicting results were found, both DFA models are determined equivalent. Otherwise, the input which causes the differing results is returned as the counterexample.

```
def equivQuery(self, hypothesis):

    n = self.noOfStates + 1
    inputs = ''.join(self.alphabet)

    for i in range(1,n):
        for combo in product(inputs, repeat=i):
            if hypothesis.accepts_input(combo) != self.dfa.accepts_input(combo):
                print("not equivalent")
                return ''.join(combo) # return counterexample

    return "equivalent"
```

Figure 3.5: Equivalence Query Code

3.3 Performance Testing

The implementation proved to be successfully able to infer DFA's after testing with automata of various state numbers. Further tests were carried out in order to measure the efficiency of the algorithm, for example detecting any bottlenecks within the algorithm and identifying how well the program would cope with larger DFA's in terms of program execution time.

Timing tests were carried out on DFA's ranging from state numbers of 2 to 20, each with 2 input symbols and random transitions. The following results were obtained.

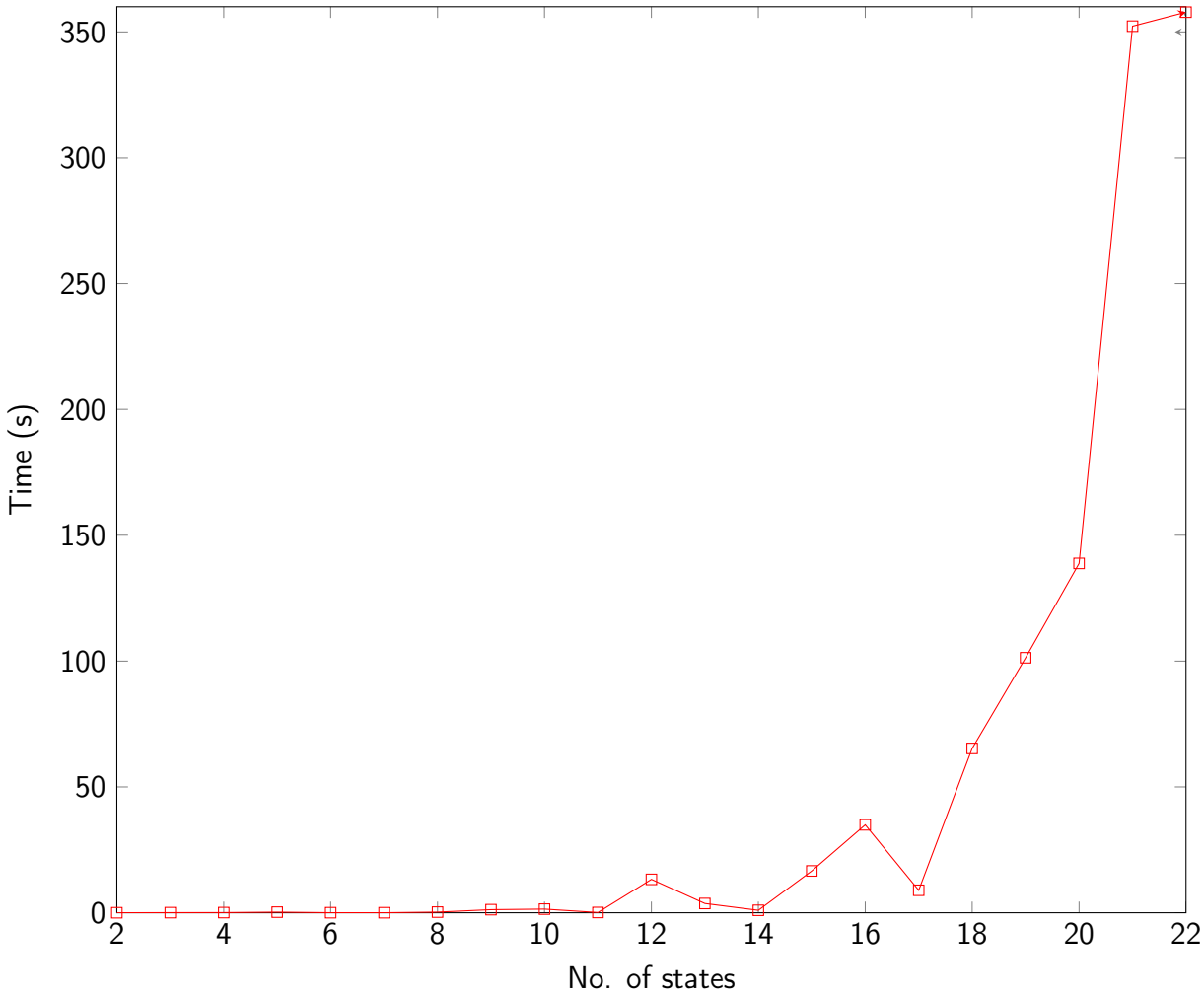

Figure 3.6: L* Execution time vs No. of States (Without Cache)

As you can see, time remains consistent from 2 states to 14, with executions of less than 1 second with a slight outlier at n = 12, with a recorded time of 13 seconds. More noticeable increases occur after this point representing an exponential increase. A difference of 30 seconds exists between state sizes 18 and 19 and again at 19 and 20. Larger margins are

seen after this point. This seems to suggest that the program doesn't scale well at larger sized DFA models, with massive jumps seen between only 1 state differences.

To investigate this, profiling was done on the program using a state size of 20 in order to identify any bottlenecks in the implementation. The python package cProfile was used for this, giving the following results:

Function	Total Time	No. of Calls
Whole Program	226s	1
Equivalence Query	150s	6
Membership Query	66s	3369434

Table 3.4: Implementation Profiling Results

As you can see, the equivalence query calls seems to take up a majority of the program execution time, with roughly 66%. The rest is taken up by membership queries, with quite a number of calls for this particular function. Upon further investigating, it seemed that many of the same membership queries were being called, thereby inflating the total number of calls. This was more common when learning larger sized DFAs. As an experiment, a **cache** was used to store past membership calls. The following results obtained with the new measure can be seen in figure 3.7:

Execution times were initially far lower with the use of a membership function cache, with an average of 63% save time, with certain points saving up to 90%. At some stage the time saved becomes negligible, with not much of a difference seen in execution times for the learning of 22 sized state DFA's with and without the use of a cache.

3.4 Possible Improvements

A number of possible enhancements can be considered to solve some of the time inefficiencies discussed in the previous section.

3.4.1 Optimisation of Membership Queries

As was observed during performance testing, membership queries were by far the most called function and this high number of calls significantly increased overall execution time. Optimisation can be carried out to decrease the average number of membership query calls. Berg et al. [11] study this, where they are successfully able to reduce membership calls using the following two observations:

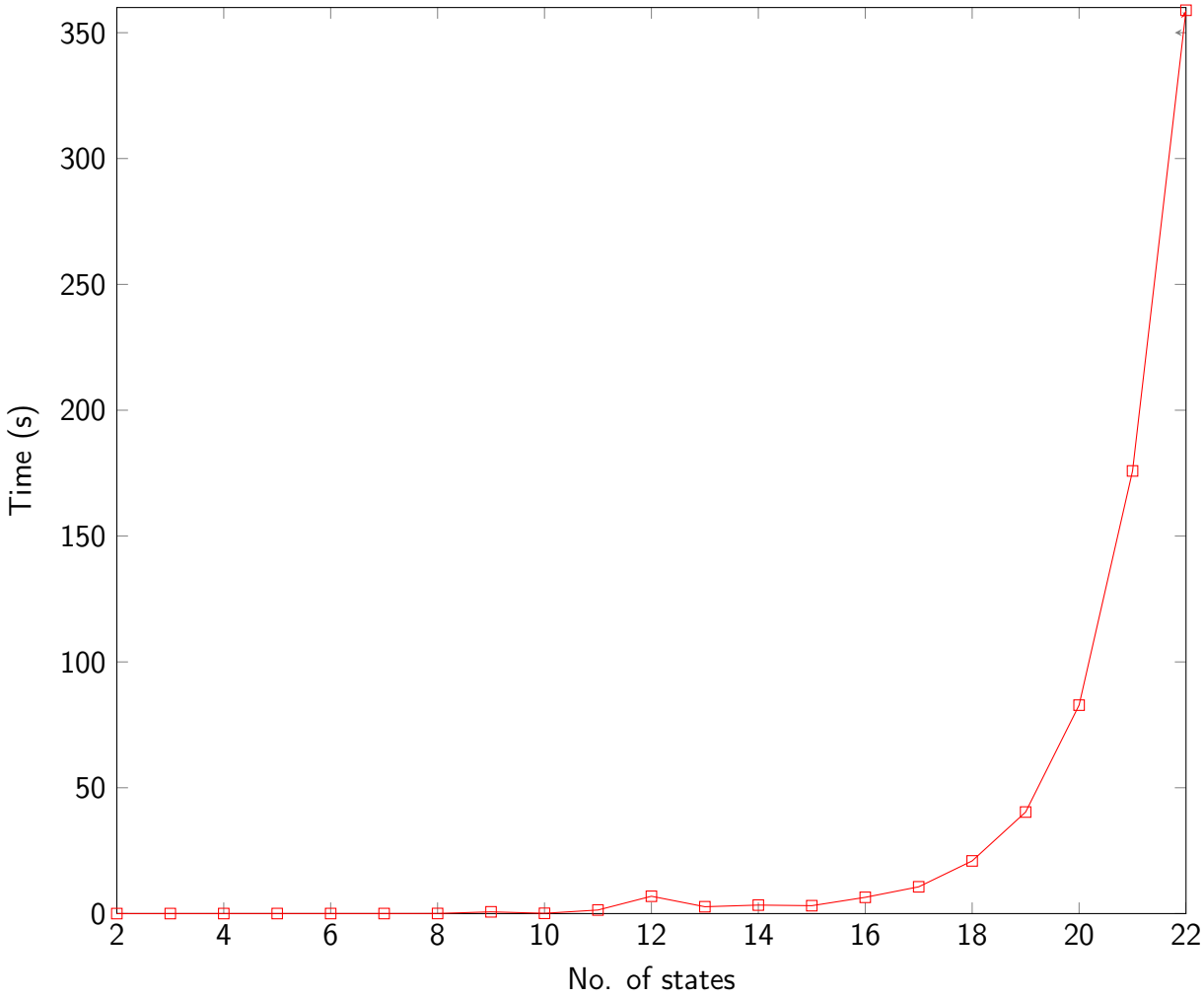

Figure 3.7: L* Execution time vs No. of States (With Cache)

- Prefixes of accepted strings are accepted.
- Suffixes of rejected strings are rejected.

Before querying a string, it can first be tested if it is an extension of any previously rejected string, following the second observation. If so, then the result is immediately added to the observation table, otherwise a membership query to the teacher is called. Berg et al. find that this feature reduces total membership query calls by 20% for randomly generated automata and 60% for real world examples, a significant improvement.

3.4.2 Faster DFA Equivalence Algorithms

For the purposes of this implementation, a brute force strategy was employed in the equivalence query function to test for the equivalence of two DFA models. This of course is not the most efficient method, with a large amount of the average execution time being spent on single equivalence query calls. Work done by Norton [21] showcase a few faster alternatives which can be used for general finite automata equivalence testing:

Symmetric Difference with Emptiness Testing

This method adopts the use of the mathematical function of symmetric difference in order to find any strings accepted by one DFA that are not accepted by the other. A third DFA named C is constructed equalling the symmetric difference of the two tested DFA, A and B, using the following equation:

$$L(C) = (L(A) - L(B)) \cup (L(B) - L(A))$$

$L(C)$ is then tested for emptiness, where $L(A)$ and $L(B)$ are equivalent if $L(C)$ does not recognise any strings.

Table Filling Algorithm

The table filling algorithm is a simple and easy to understand strategy to equate two DFAs. The states of both DFA are looked at to identify which states are distinguishable at some input. As these distinguishable states are found, they can used to find more distinguishable states, where this information is continually being stored in a table. At the end of the algorithm, if the two start states of the algorithm are found to be not distinguishable, then the DFAs can be determined equivalent. Time complexity of this algorithm is $\mathcal{O}(n^2)$.

Hopcroft Algorithm

Hopcroft devised an algorithm of time $\mathcal{O}(n \log n)$ (n being the total number of states being considered) which can minimise DFA and determine DFA equivalence, in 1971 [22]. The algorithm systematically partitions states into groups until no more partitioning can be done, where equivalence can be determined by viewing the location of the start states in these groups. The actual algorithm is much more complex and it is advised to view Hopcroft's original paper for a better understanding.

3.4.3 Practical Equivalence Queries

So far we have discussed how equivalence queries test the equivalence between a learned hypothesis and the target DFA, either accepting or returning a counterexample. While this works for simple models such as the implementation described above, this may not be exactly feasible in real life black box scenarios. Steffen et al. [23] discuss how in practical situations, one must resort to approximate solutions of equivalence queries, which are typically based on membership queries, with an efficient solution based on this discussed in [24] which only required 3 or 4 membership queries.

3.5 Conclusion

In this chapter we gave an introduction the L* algorithm, an automata learning algorithm for deterministic finite automata. This included a basic description of the algorithm as well as an implementation based on it. This gave an insight into the overall principals of automata learning algorithms, particularly the MAT model, which becomes relevant when discussing approaches of other algorithms discussed in subsequent chapters.

4 Tomte (Register Automata Learning)

The relevant work chapter briefly looked at a few examples of register automata learning techniques. One of the more notable ones was discussed in a paper [1] by researchers at Radbound University who have also implemented this approach in their own research tool called Tomte, which is utilised in the FRA learning approach described in this dissertation. This chapter will discuss the workings of Tomte [9] and how it fits in the overall approach of FRA learning.

4.1 Tomte Design

Tomte follows the same concept of many automata learning algorithms that have come before it, all deriving from Angluin's original work [2]. This is the already mentioned MAT model, where the learning process consists of a learner and a teacher, where the learner infers the unknown automata from the teacher by asking different types of queries. These are typically membership and equivalence queries, though some implementations such as this one ask *output* queries, which returns the expected outputs for a concrete sequence of inputs. This type of automata is usually only present when learning automata with output symbols such as Mealy machines or register automata which return output. Learning algorithms make use of these queries to create a sequence of automata which converge to the correct automata. This is effectively expressed in the previous chapter when describing Angluin's L* algorithm. This process is also covered in detail in [25].

Fig 4.1 presents the overall architecture of Tomte as described in the register automata learning paper [1]. Five main components make up the system. On the right we can see the *teacher* or the *system under learning*, i.e. an implementation of a register automata we wish to learn. At the opposite end we can see the *learner*, which in this system is a Mealy machine learning tool. The specific tool Tomte uses here is LearnLib [26], which implements many different learning techniques for both DFAs and Mealy machines.

Three components exist between the learner and the teacher. These are the *abstractor*, the *lookahead oracle* and the *determiniser*. Each of these play a key role in the working of this system:

- **Determiniser**: The role of the determiniser is to eliminate the non-determinism of the SUL, which can occur due to the presence of fresh output values in some register automata, where different output values may be generated in different runs of the automata. While this feature is ideal for modelling real world systems such as a login system which can generate user passwords, LearnLib only has the ability to learn deterministic systems. The determiniser eliminates this non-determinism by exploiting symmetries in the automata which are captured through automorphism. As this is out of scope for this particular dissertation, the reader is advised to view a more comprehensive explanation found in the mentioned paper [1].
- **Lookahead Oracle**: The task of the lookahead oracle is to annotate each output action of the SUL with a set of values that are *memorable* after the action occurs. A data value is memorable if it has any effect in the future of the automata, for example if it occurs in a future output or if a future output depends on any equality between this data value and and a future input parameter. This is essential to the effective learning of any such data values in the learned register automata. The lookahead oracle keeps track of these values by storing traces in an *observation tree*, which also acts as a cache for any repeated queries to the teacher.
- **Abstractor**: Arguably the most significant component of the system, the abstractor maps the concrete symbols (input and output) of the SUL to abstracted symbols which can be handled by the Mealy machine learner, essentially reducing the task to learning a Mealy machine with an abstract alphabet. These mappings are done via a series of implemented mappers in Tomte. Due to the complex nature of these mappers, the reader is advised to view [1] for a more detailed explanation.

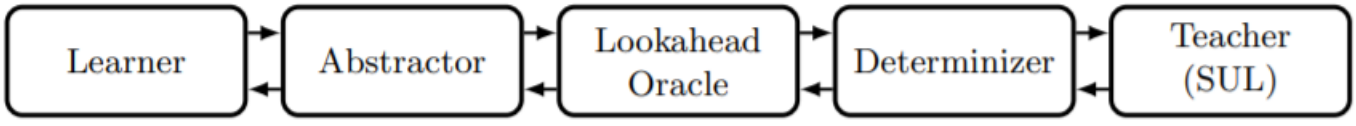

Figure 4.1: Architecture of Tomte (diagram from [1])

4.2 How Tomte Was Used

The overall goal of this research was to use existing learning techniques to learn fresh-register automata (FRA). Due to the similarity between these and register automata, it was proposed to make use of register automata (RA) learning tools to carry this objective out.

Tomte would act as the main learning tool within the overall architecture of the FRA learning system which required the learning of RA models (described in more detail in section 5.1). Tomte would read an input file describing a register automaton in an XML file format, along with other required parameter files. Learning can then be initiated, where a 'learned' register automata file in the same format as the input file, is outputted by Tomte.

4.2.1 Model File

The register automata model is described in an XML file format specifically to be read by Tomte. Different tags in the XML file are used to specify different parts of the RA model. For example, a declaration tag is used to list the global actions (inputs) and registers that are used within this RA model. After this, there are location tags for each different state in the automaton, before a transition tag to describe each different transition in the model. Within this transition, there is a source and target, as well as a label tag to specify the action, guard(s) and assignment(s) for that particular transition. Tomte's format requires there to be two transition tags for every transition in the RA being modelled, with one for the input symbol/action followed by a transition just specifying the output symbol produced. These XML model files incorporate all the different components described in the register automata definition (see chapter 2). Fig. 4.2 shows an example of this model file describing the "stack" in Fig. 4.3.

```
<declaration>// Place global declarations here.
int r1; // registers
int r2;
void IPut(int p) { //input actions
}
void IGet() {
}
void OOK() {   //output actions
}
void OOut(int x) {
}

...

<location id="id0"/>
<location id="id1"/>
<location id="id2"/>
<location id="id3"/>

...

<transition>
 <source ref="id7" />
 <target ref="id2" />
 <label kind="synchronisation">IGet()</label>
</transition>
<transition>
 <source ref="id2" />
 <target ref="id8" />
 <label kind="synchronisation">OOut(x1)</label>
</transition>

...
```

Figure 4.2: Abbreviated RA Model File of a Stack

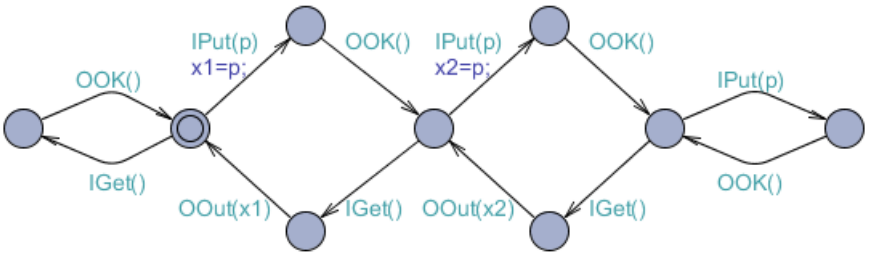

Figure 4.3: RA Model of Stack

4.2.2 SUT Information File

Another one of the files Tomte makes use of in the learning process is a SUT information file which relates directly to the input model file. This file specifies the list of symbols, both input and output that are used in the SUL. The learner uses this information, which follows the concept described in the previous chapter where the only information the learner has is the alphabet of the automata being learned. These are defined in a YAML file as seen in Figure 4.4:

```
constants: []
inputInterfaces:
  IPut:
  - Integer
  IGet: []
name: Stack
outputInterfaces:
  OOK: []
  OOut:
  - Integer
```

Figure 4.4: SUT Information File

4.2.3 Configuration File

Before running Tomte, a configuration file specifying learner parameters must be created by the user. These parameters are needed for generating a hypothesis during learning. These learning parameters include the name of the SUT file which contains the alphabet of the SUT which the learner requires, as well as a seed value to implement pseudo randomness in the learner. Testing parameters are also specified here, which specify the kind and number of test traces used to test the hypothesis.

```
learning:
    sutinfoFile: "sutinfo.yaml"
    seed: 1299777356020

testing:
    minValue: 256
    maxValue: 511
    minTraceLength: 100
    maxTraceLength: 100
    maxNumTraces: 1000
    resetProbability: 0.05
```

Figure 4.5: Example Config File

4.3 Challenges/Observations of Tomte

With the use of Tomte throughout this research project, a number of observations and challenges were discovered, which provided some difficulty in the scope of the entire system.

4.3.1 Minimisation of Learned Automata

A feature of most automata learning algorithms, Tomte included, was the fact that that algorithm will always return a minimal automaton. An automaton is minimal if there exists no equivalent automaton with strictly less states. This means that the automata model learned by Tomte is not strictly the same as the SUL if the SUL was not minimal to begin with, though the language shared between the automaton is the same. A minimal automata produced by Tomte may not include states and transitions present in the SUL automata model, as well as other features such as guards, assignments and registers. While this feature is ideal in general automata learning, it proves to be a hindrance in the grand scheme of our FRA learning system, which is discussed in more detail in the next chapter.

4.3.2 Input Complete Necessity

Tomte requires the models learned to be input complete, meaning a transition must be specified for every input for an input state. If these are not specified, then the SUL which simulates the model will assume that these transitions exist with a transition looping back to the same states with the output symbol "Oquisience". The prevents the need to alert the user of an error stating the input model is not input-complete. This can be an issue as the learned model can contain these extra transitions and states if the original SUL is not input complete, which makes it difficult to exactly match the input model with the learned model.

5 Learning Fresh-Register Automata (Attempt 1)

At the moment, π-calculus models are geared mainly towards model checking, with no current techniques to our knowledge of learning these models. However, a new kind of automata known as fresh-register automata (FRA) [7] have been shown to encode the behaviour of π-calculus processes. With this knowledge, we attempt to use techniques of automata learning to learn FRA models, which in turn would allow for successful learning of π-calculus models. This chapter looks at the first attempt at implementing such a system.

5.1 FRA Learning Design

As has been mentioned throughout this dissertation, the main aim is to make use of register automata learning techniques in order to learn fresh-register automata (FRA). The general method proposed to do this is as follows:

1. The FRA model to be learned would be mapped to a RA equivalent using a predefined set of mappings.
2. The converted RA model would be learned using the register automata learning tool Tomte [9].
3. The learned RA model would then be mapped back to a fresh-register automata.

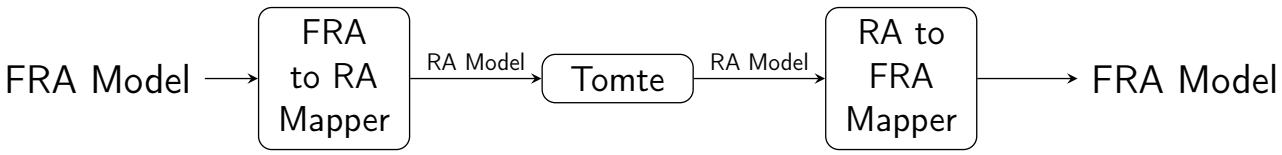

Figure 5.1: FRA Learning Architecture

This architecture required the implementation of two main components, the FRA to RA mapper and the RA to FRA mapper. These are typical programs which can read an FRA and RA model file respectively and convert it to its counterpart. The main hurdle within this project would be to come up with predefined mappings which would be used for the conversion from fresh-register automata to register automata and vice versa. The rest of this chapter will look at the mappings defined as well the different implementations in the learning architecture.

5.2 Mappings

Before implementing the automata converters/mappers, mappings between register automata input symbols and fresh-register automata input symbols first needed to be defined. The idea was to make use of register automata components which could easily be mapped to FRA inputs. These include the use of registers involved in register automata inputs which would match the input of an FRA. These registers would be a part of either the guard or assignment statement in an RA transition, depending on the FRA transition being mapped. The following table shows the general mappings that were used in this implementation. Included are the equivalent π-calculus transitions to give an overall view of the system.

π-calculus	FRA	RA
Known Input Transition	i	Act(x), i==x
Known Output Transition	i'	ActOut(x) i==x
Locally Fresh Input	i•	ActDot(x), i:=x;
Globally Fresh Input	i⊛	ActStar(x) i:=x;

Table 5.1: Automata Mappings

As you can see, different register automata actions are associated with each type of transition in an FRA. Each RA action is written to be self explanatory, for example an i• is mapped to the action name *ActDot* and i' to the action name *ActOut* to represent a simple output transition in an FRA. While these actions are sufficient to specify the type of FRA

transition, extra information on the RA side is needed to indicate the register being used in the FRA transition. This is done with the use of guards and assignments, with the register number used here matching the one used in the FRA transition.

The use of a guard or an assignment is chosen to closely match the logic of the FRA transition. For example, an FRA *known* transition requires the specific register to already contain the input before the FRA accepts said input. This logic can be seen replicated in the RA mapping, with a guard label checking for equality between the input and the register in question. It should be noted that the specific mapping logic chosen here has no bearing on the learning of the RA model and was only chosen to show a more obvious conversion from FRA to RA to the user. In reality, any mapping logic regarding register assignments and guards would be suitable here as long as the register number is clearly indicated in the RA transition.

The following are examples of FRA transitions and their RA equivalents:

FRA	RA
1	Act(x) r1 == x
2'	ActOut(x) r2 == x
3●	ActDot(x) r3 := x
3⊛	ActStar(x) r3 := x

Table 5.2: Example of Mappings

When mapping back from an RA to an FRA, the two pieces of information that will be looked at in the register automata transitions are the action name and the assignment/guard. The action name allows for the type of FRA transition to be identified, while the register number involved in the guard or assignment will have the same effect for the FRA. The following figures are examples of equivalent register automata and fresh-register automata using these mappings:

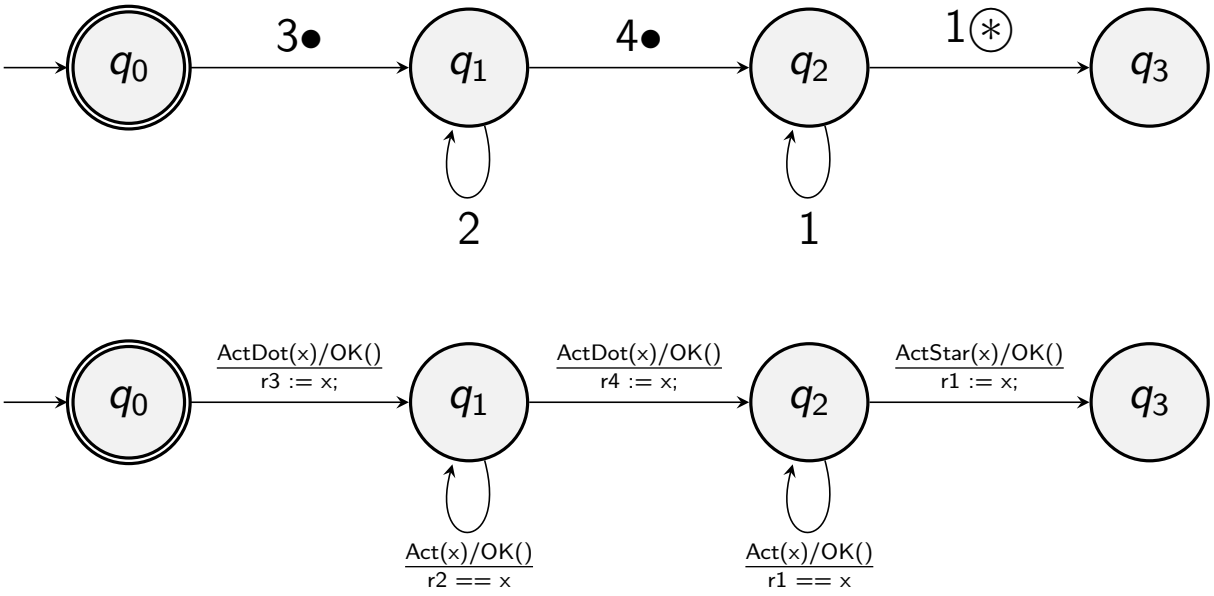

Figure 5.2: Example of a simple FRA and its RA counterpart using the above defined mappings.

5.3 Implementation

5.3.1 FRA to RA Mapper

The first component of the FRA learning architecture is a mapper to convert the FRA model to be learned into a register automata which would be learned via Tomte. After defining the mappings to do this, a mapper was created to read an FRA model and then create the register automata model file readable by Tomte (see section 4.2.1).

For the purposes of this project, the mapper reads FRA descriptions defined in 'dot' files, which are the text file format of the suite Graphviz [27]. The following figure is an example of a Graphviz description of the FRA from Figure 5.2.

The mapper is programmed in the programming language Python. A library known as pydot is first used to parse the readable dot file. This is done to extract the list of states and the transitions between these states. These are stored in a list and a dictionary respectively. The dictionary is then iterated where the current transition is then written into a new XML file according to Tomte's model file standards. The same is done for the states. This is done using Python's ElementTree XML API. Figure 5.4 is an example of a transition being written into an XML file if the FRA transition happens to be a locally fresh or (•) transition.

An important thing to note is most FRA models of π-calculus systems have double inputs, as can be seen in Figure 5.4. How this is translated to an RA model using this mapper is by

```
digraph {
    q0 []
    q1 []
    q2 []
    q3 []


   q0 -> q1 [label="3."]
   q1 -> q2 [label="4."]
   q1 -> q1 [label="2"]
   q2 -> q3 [label="1*"]
   q2 -> q2 [label="1"]
}
```

Figure 5.3: FRA Definition

creating a double transition. The edge for the first input points directly to a state which outputs a transition for the second input, before this transitions to the target state.

Due to restrictions with Tomte, some extra transitions needed to be included in the register automata model.

- Tomte requires every state to be the source of at least one transition. There may exist cases in FRA models where this may not be true. Such an example can be seen in figure 5.2 with state q3 not containing any outward transitions. This problem is solved in this implementation by creating a "dummy" self loop for these states in the RA model. For example, q3 would have a transition pointing to itself with the action *ActDummy()/OK()*. This would be done for all states fulfilling this condition. When mapping back from RA to FRA, "dummy" transitions will not be taken into account, giving the illusion it did not exist in learning.
- Tomte does not assume registers are initialised at the start of a run. An RA model where an *Act()* or *ActOut()* can occur before *ActDot()* or *ActStar()* can not be learned by Tomte due to errors occurring from the guard checks in the former two actions. A register and an input cannot be checked for equality if nothing has been stored in the register beforehand. The solution to this problem was to add a transition at the starting state, which assigned all the registers used in the FRA with a value. The action in this transition would be *ActLoad()*.

```
transition= SubElement(template, 'transition')
source = SubElement(transition, 'source', ref= "id" + str(location_count))
location_count += 1
target = SubElement(transition, 'target' , ref = "id" + str(location_count))
label_sync = SubElement(transition, 'label', kind="synchronisation",x='0',y='0')
label_sync.text = "IActDot(x)"
label_assign = SubElement(transition, 'label', kind="assignment", x="0", y="0")
label_assign.text = "r"+number1+"=x;"

transition = SubElement(template, 'transition')
source = SubElement(transition, 'source', ref="id" + str(location_count))
location_count += 1
target = SubElement(transition, 'target' , ref = "id" + str(location_count))
label_sync = SubElement(transition, 'label', kind="synchronisation",x='0',y='0')
label_sync.text = "OOK()"
```

Figure 5.4: FRA to RA Mapper Snippet

5.3.2 RA to FRA Mapper

After Tomte outputs the learned register automata model, it is converted to an FRA using a separate mapper program. After this we can observe if we have successfully learned the initial FRA model.

The mapper developed for this is somewhat similar to the previous mapper. In this case, the learned register automata model file is parsed, extracting all the states and transitions. These states and transitions are written into a dot file to be shown in Graphviz, akin to Figure 5.3. The edge labels are written depending on the transition in the learned model file. For example, if a transition in the learned file is *ActDot(p0)* with *x3 := p0*, then the edge label in the FRA version will be written as 3•.

```
for k,v in transitions.items():
    dot.node(k)

for k,v in transitions.items():
    for k2,v2 in v.items():
        dot.edge(k,v2,label=k2)
```

Figure 5.5: Writing FRA

5.4 Evaluation

A number of known FRA models were tested using this learning architecture, with varying results.

The main observation from these trials was that Tomte was not learning all the assignments and guards from the SUL register automata model, and in some cases registers were being omitted in the learned model. This was detrimental as it would prevent effective mapping back from the RA to the FRA. The following is an example of this situation occurring in the FRA model of a ping system originally modelled in π-calculus:

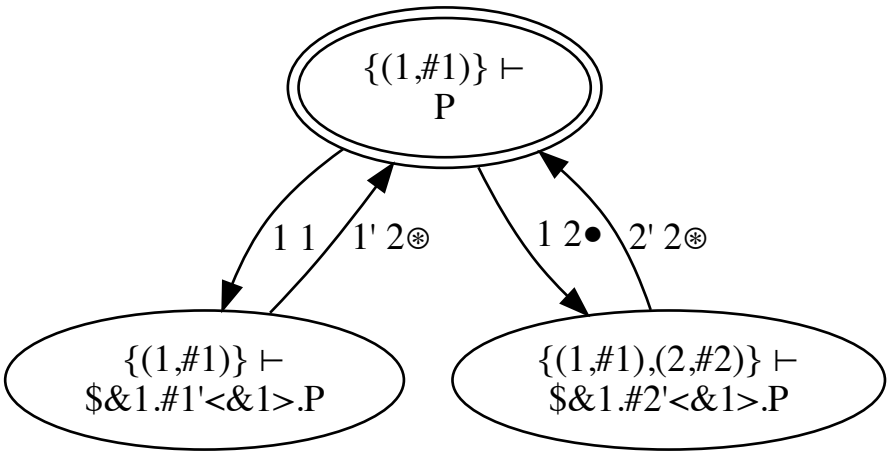

Figure 5.6: FRA Model of a Ping System

The above model is first converted to a register automata to be learned by Tomte. Using the already mentioned mappings and conversion details, this corresponds to:

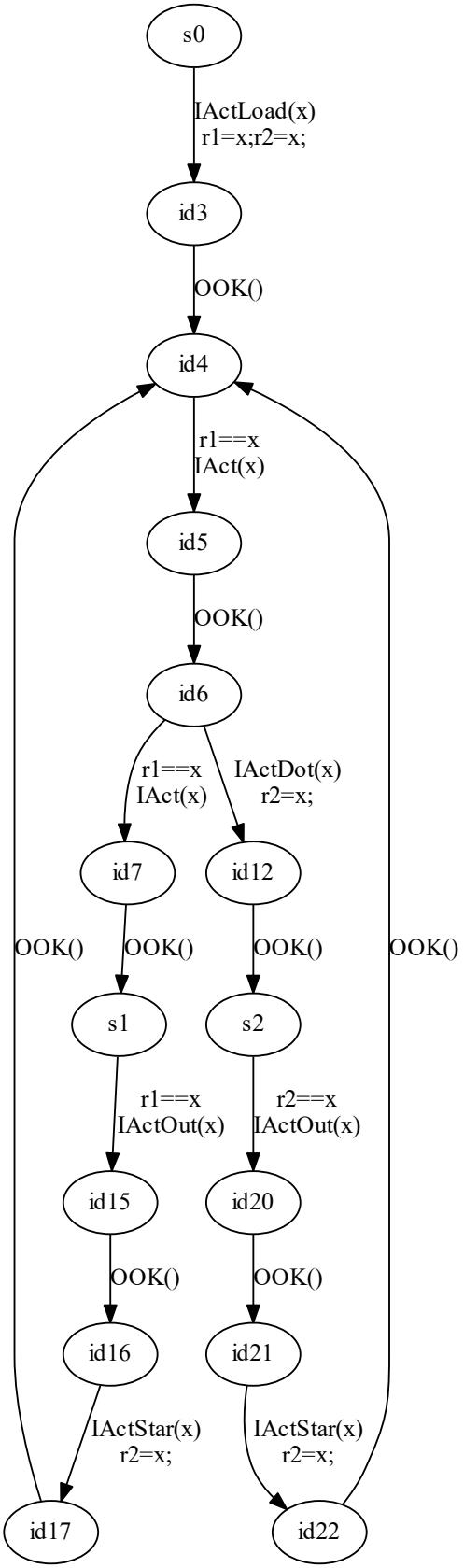

Figure 5.7: Ping System Converted to RA (Visualised Using Graphviz)

This model is then learned using Tomte, which outputs the following learned RA model:

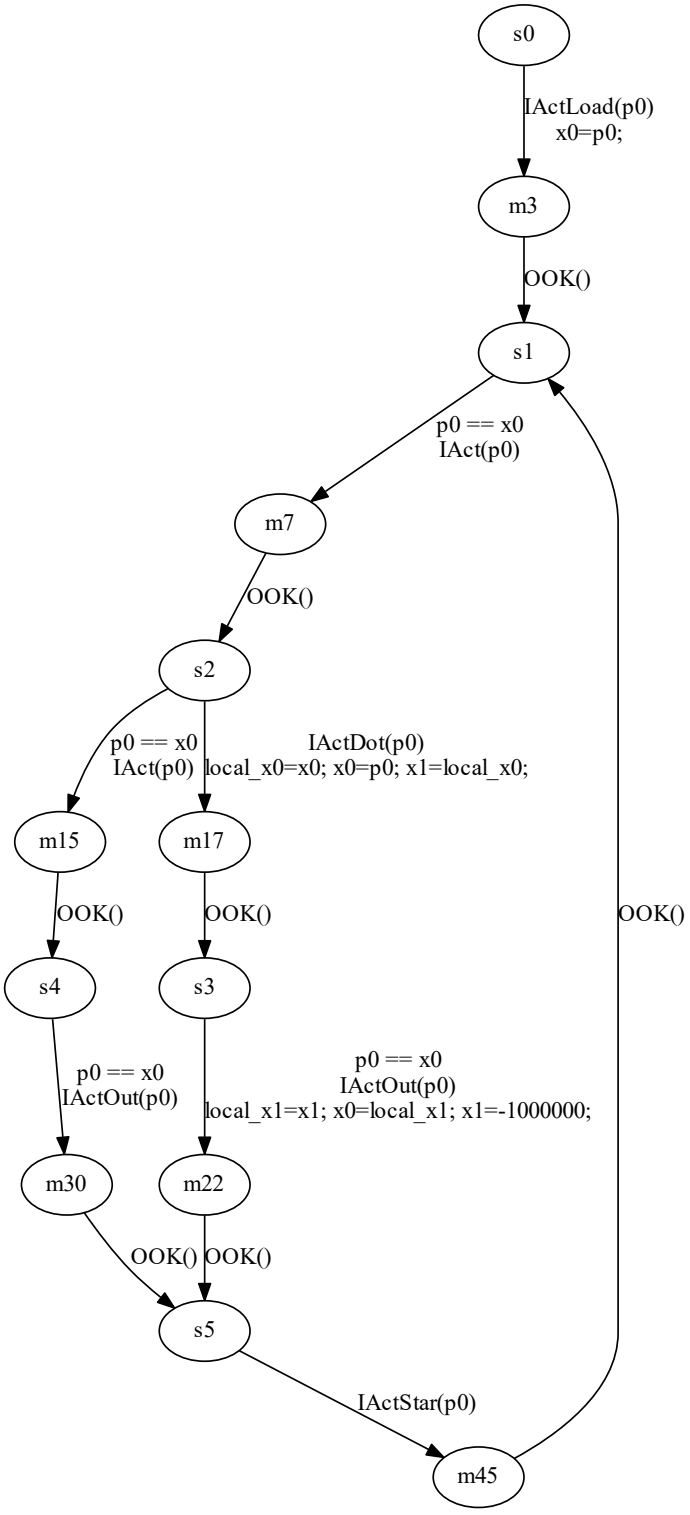

Figure 5.8: Learned RA Ping System

Note in both of the above models that Tomte merges transitions which are the identical. For example, in the original FRA example (Fig 5.6), both of the transitions out of the starting state begin with '1'. In the RA versions, this transition is shared before it diverges into the other two states with different input symbols. This is also done for the '2⊛' pointing back to the starting state.

For the most part, the learned model seems to match the SUL. However, the key difference which prevents effective mapping back to an FRA is the lack of an assignment statement in the *ActStar()* which points back to the starting state. This prevents us from verifying whether the register that was involved here in the original model is register "1" or "2", and thus we cannot map this to an FRA.

This can be explained by Tomte's tendency to learn the *minimal* version of automata. As this particular statement has no significance on the operation of the rest of the automata i.e. traces can't differ based on this statement, then Tomte does not include it in the learned model. As models being learned become larger, these sorts of situations become more numerous, making it even more difficult if not impossible to properly map back to an FRA model.

Other observations from the observed model include the *IActDot* transition and the *IAct* transition which follows. In the original RA model, this action was accompanied with the assignment of the input to the second register. What Tomte instead does is move the current contents of the first register to the second register and store the input into the first register. For an similar reason, this also happens in the next transition but with the registers switched and with an arbitrary number of -100000 stored in the second register.

5.5 Conclusion

All of these inconsistencies between the learned model and the original model prevent effective mapping back to the FRA and hinder the ability to effectively learn FRA models using this architecture. From this we can conclude that register numbers are not an effective way to differentiate between types of FRA inputs. As registers can be considered an internal component of the system, models which are equivalent may use different register numbers in different circumstances. With this is mind, new mappings were defined which did not rely on register assignments or guards in an attempt to enforce effective mapping between RAs and FRAs. This new attempt is discussed in the following chapter.

6 Learning Fresh-Register Automata (Attempt 2)

Due to the inconsistent results obtained from the first FRA learning approach, a second attempt was made hoping to eliminate the problems existing with the first attempt. The main differences made in this approach were a change in FRA-RA mappings, primarily due to this aspect being the main source of error in the previous attempt.

6.1 FRA Learning Design

The learning architecture remains the same in this approach, in terms of the use of mappers to convert an FRA to RA and vice versa, as well as the use of the Tomte learning tool.

6.2 Mappings

In the previous approach, information of each FRA transition was represented as every component of an RA transition. These components were the action, which represented the type of FRA transition, the guard and the assignment, which indicated what register number was in use in the FRA transition. We concluded that the use of assignments/guards to represent the register number was not ideal due to Tomte's inconsistency in learning these, as well as the fact that equivalent models can differentiate on register numbers due to their internal nature.

From this, it was decided to store all the information of an FRA transition into the action portion of an RA, doing away with the assignment/guards and registers entirely. This would involve having a separate RA action for each different FRA input. The register number in the FRA input would be observed in the RA action name, as the following table highlights:

π-calculus	FRA	RA
Known Input Transition	i	Act**i**()
Known Output Transition	i'	ActOut**i**()
Locally Fresh Input	i•	ActDot**i**()
Globally Fresh Input	i⊛	ActStar**i**()

Table 6.1: Automata Mappings

Due to the action being the only part of each transition, the mapped model is reduced to a general DFA. One drawback that was already forecasted with this method was the excessive amount of possible inputs in the automata if the original FRA being learned contained a high number of registers. For each register in the FRA, 4 different RA inputs would exist, representing each type of FRA transition as seen in the table above.

FRA	RA
1	Act1(x)
2'	ActOut2(x)
3•	ActDot3(x)
3⊛	ActStar3(x)

Table 6.2: Example of Mappings

6.3 Implementation

The implementation involved for this approach was almost identical to the implementation in the first approach. Both an FRA-to-RA and RA-to-FRA mapper was developed, with the only differences being the mappings that were taken into account when parsing and writing into model files.

6.4 Evaluation

As with the previous approach, multiple FRA models were tested to be learned using this learning architecture, with mixed results.

The FRA example from figure 5.6 was successfully learned using this new set of mappings. All the transitions in the original RA were successfully present in the learned RA model with the correct actions, allowing the mapper to effectively map the model back to an FRA which was equivalent to figure 5.6.

While the new set of mappings solved the issue of registers/assignments/guards not being learned correctly which was present in the old mappings, there were still instances of the learned RA models being minimal. This is apparent in the following example.

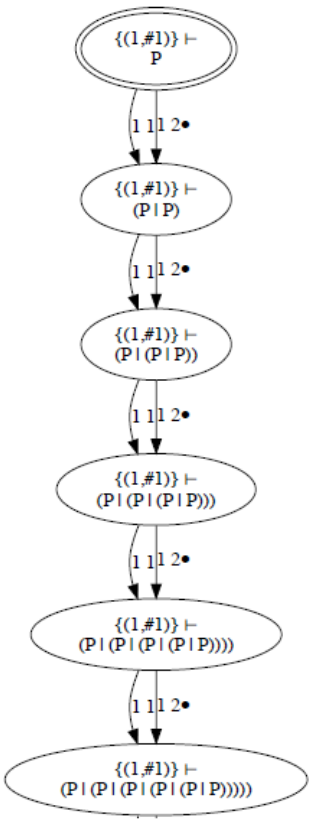

Figure 6.1: FRA Example

The above FRA example was converted to a register automata model file using the set new of mappings before subsequently being learned by Tomte. The learned RA mapped back to FRA as follows:

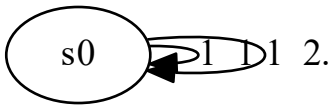

Figure 6.2: 'Learned' FRA

As you can see in the above learned FRA, only one state is present compare to six in the original, with the actions pointing to itself. Tomte learns the minimal version of the original automaton with both of these automata being equivalent (share the same language). Similarly to the last approach, this feature prevents the effective learning of certain FRA models, though the success rate of this approach is noticeably higher than that of attempt 1.

The major downside with this logic of mappings is the far greater number of input symbols involved, with four different actions for each register involved in the RA model. This can lead to performance issues with larger models, due to the need for more membership queries to be asked by the learner to the teacher during the learning process.

6.5 Conclusion

Despite the slightly more promising results of this set of mappings compared to the first attempt, it still did not manage to consistently and effectively learn all tested fresh-register automata , mainly due to the minimisation problem described above.

7 Conclusion

The main objective of this dissertation was to investigate whether fresh-register automata (FRA) could be learned/inferred. As these automata can encode the behaviour of π-calculus models, this investigation essentially extends to whether these models can be learned. This was done with the development of a learning architecture, where the FRA model to be learned would be mapped, using a predefined set of mappings, to a register automata model, before being learned with the learning tool Tomte, and then finally be mapped back to an FRA model.

From the two different attempts to define mappings and implement this approach, it was concluded that FRA models could not be consistently learned via the use of the Tomte tool.

Attempt one, which made use of the full components of the register automata model including action names, registers and guard/assignment statements, in its mappings between FRA and RA, saw general cases where the learned RA model would not exactly match the input RA model, as it contained fewer states and transition information. This was due to a feature of Tomte where the learned model is the minimal version of the input model. This prevented the effective mapping of some learned RA back to an FRA due to the lack of learned information needed to properly map from the RA model to an FRA model.

Attempt two redefined the mappings, using only the action name of a register automaton state transition when mapping from a FRA state transition. This allowed for the successful mapping from a learned RA model to a FRA model, with cases of the mapped FRA model matching the original FRA model, indicating successful FRA learning. Despite this success, there existed examples where the learned FRA did not match the original model. This again was due to Tomte's minimisation feature, where the minimal learned model would sometimes contain less states and transitions compared to the original model, preventing the successful learning of an FRA.

7.1 Future Work

The main work that can be done is developing a solution regarding the mapping problem that was described above. The preference would be to develop a learning system that can work with the first set of mappings, due to the already mentioned issues with Tomte. This could involve developing a custom register automata learning algorithm and tool that learns the exact SUL model, although this by itself would be a significant research project.

Other work that could be done is the development of an FRA equivalence checking tool, which could verify whether the learned FRA model is equivalent to the FRA model under learning.

Bibliography

[1] Fides Aarts, Paul Fiterau-Brostean, Harco Kuppens, and Frits Vaandrager. Learning register automata with fresh value generation. In Martin Leucker, Camilo Rueda, and Frank D. Valencia, editors, *Theoretical Aspects of Computing - ICTAC 2015*, pages 165–183. Springer International Publishing, 2015.

[2] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87 – 106, 1987. ISSN 0890-5401.

[3] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. 01 2001. ISBN 978-0-262-03270-4.

[4] Fides Aarts, Joeri Ruiter, and Erik Poll. Formal models of bank cards for free. pages 461–468, 03 2013. ISBN 978-1-4799-1324-4. doi: 10.1109/ICSTW.2013.60.

[5] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 193–206. USENIX Association, 2015. ISBN 9781931971232.

[6] Paul Fiterău-Broștean, Ramon Janssen, and Frits Vaandrager. Combining model learning and model checking to analyze tcp implementations. In *Computer Aided Verification*, pages 454–471. Springer International Publishing, 2016.

[7] Nikos Tzevelekos. Fresh-register automata. volume 46, pages 295–306, 01 2011. doi: 10.1145/1926385.1926420.

[8] Robin Milner. Communicating and mobile systems - the pi-calculus. 1999.

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

[10] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–370. Springer Berlin Heidelberg, 2002.

[11] Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena. Insights to angluin's learning. *Electronic Notes in Theoretical Computer Science*, 118:3 – 18, 2005. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2004.12.015. Proceedings of the International Workshop on Software Verification and Validation (SVV 2003).

[12] Malte Isberner, Falk Howar, and Bernhard Steffen. Learning register automata: From languages to program structures. *Machine Learning*, 96:65–98, 07 2013. doi: 10.1007/s10994-013-5419-7.

[13] Oliver Niese. An integrated approach to testing complex systems. 2003.

[14] Muzammil Shahbaz and Roland Groz. Inferring mealy machines. volume 5850, pages 207–222, 11 2009. doi: 10.1007/978-3-642-05089-3_14.

[15] Muhammad Irfan, Roland Groz, and Catherine Oriat. Improving model inference of black box components having large input test set. 09 2012.

[16] Sofia Cassel, Falk Howar, Bengt Jonsson, Maik Merten, and Bernhard Steffen. A succinct canonical register automaton model. *J. Log. Algebraic Methods Program.*, 84: 54–66, 2015.

[17] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring canonical register automata. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 251–266. Springer Berlin Heidelberg, 2012.

[18] Frits Vaandrager. Active learning of extended finite state machines. In Brian Nielsen and Carsten Weise, editors, *Testing Software and Systems*, pages 5–7. Springer Berlin Heidelberg, 2012.

[19] Yi Xiao and Emilio Tuosto. On learning nominal automata with binders. *Electronic Proceedings in Theoretical Computer Science*, 304:137–155, Sep 2019. ISSN 2075-2180. doi: 10.4204/eptcs.304.9. URL `http://dx.doi.org/10.4204/EPTCS.304.9`.

[20] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, 2009. ISBN 1441412697.

[21] Daphne Norton. Algorithms for testing equivalence of finite automata, with a grading tool for jflap. 2009.

[22] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*, pages 189 – 196. Academic Press, 1971. ISBN 978-0-12-417750-5. doi: https://doi.org/10.1016/B978-0-12-417750-5.50022-1.

[23] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. pages 256–296, 06 2011. doi: 10.1007/978-3-642-21455-4_8.

[24] Falk Howar, Bernhard Steffen, and Maik Merten. From zulu to rers - lessons learned in the zulu challenge. pages 687–704, 10 2010. doi: 10.1007/978-3-642-16558-0_55.

[25] Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. pages 256–296, 06 2011. doi: 10.1007/978-3-642-21455-4_8.

[26] Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source learnlib: A framework for active automata learning. 07 2015. doi: 10.1007/978-3-319-21690-4_32.

[27] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.