KTH Computer Science
and Communication

Incremental Learning and Testing of Reactive Systems

MUDDASSAR AZAM SINDHU

Licentiate Thesis
Stockholm, Sweden 2011

TRITA-CSC-A 2011:14
ISSN 1653-5723
ISRN KTH/CSC/A–11/14-SE
ISBN 978-91-7501-062-5

KTH CSC
SE-100 44 Stockholm
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framläges till offentlig granskning för avläggande av teknologie licentiatexamen i datalogi Fredagen den 30 September 2011 klockan 10.00 i K2, Kungl Tekniska högskolan, Teknikringen 28 , Stockholm.

Tryck: Universitetsservice US AB

Dedication

To

Ammi, Abbu and Fari

Abstract

This thesis concerns the design, implementation and evaluation of a specification based testing architecture for reactive systems using the paradigm of *learning-based testing*. As part of this work we have designed, verified and implemented new incremental learning algorithms for DFA and Kripke structures. These have been integrated with the NuSMV model checker to give a new learning-based testing architecture. We have evaluated our architecture on case studies and shown that the method is effective.

Acknowledgements

I owe my deepest gratitude to my supervisor Prof. Karl Meinke without whose support this thesis would not have been possible. His thorough understanding of the subject, deep insight into the field and expertise during his lectures and meetings was always a source of knowledge and inspiration for me. His apt comments and constructive criticism always helped me to improve my work. His patience and encouragement whenever I was down during the last two and a half years always enabled me to re-start with a new zeal. I feel honoured and proud that he is my supervisor.

I want to thank the Higher Education Commission of Pakistan which gave me the scholarship for pursuing my PhD and also Quaid i Azam University, Islamabad, Pakistan which allowed me to come to Sweden on study leave despite a shortage of faculty in its Computer Science department.

I also want to thank two great scientists in the field of model checking which are Gerard J. Holzmann (at NASA Laboratory for Reliable Software) and Prof Mordechai Ben Ari (at Weizmann Institute of Science, Israel) who always gave prompt responses to my emails whenever I sent queries to them during my "struggle" to integrate a model checker with our testing framework.

I want to thank my office mates Niu Fei and Andreas Lundblad with whom I shared several light moments in the office. Both of them were always there to give a comment or two whenever I asked for. Especially Andreas always provided very constructive criticism of my Java code and often suggested useful tweaks to fine tune the code.

I thank Dilian Gurov my first teacher at KTH who introduced me into the world of formal methods and model checking. The courses taught by him enabled me to set the foundation for this thesis. Apart from the courses I also learnt several intricacies in the art of teaching from him, he might not know this though.

Finally I would like to mention my beloved parents, wife and family members. I don't need to thank them because I know their prayers are always with me and I am what I am because of them and their prayers.

Contents

Contents **ix**

I Introduction and Literature Survey **1**

1 Introduction and Background to Software Testing **3**
1.1 Introduction . . . 3
1.2 Specification Based Testing . . . 4
1.3 Black Box and Glass Box Testing . . . 5
1.4 Conformance Testing . . . 6
1.5 Model Based Testing . . . 7
1.6 Learning Based Testing . . . 8
1.7 Inductive Testing . . . 9
1.8 Static Checking . . . 9
1.9 Full Verification . . . 11

2 Principles of Finite Automata **13**
2.1 State Machines and Formal Languages . . . 13
2.2 Reactive Systems . . . 16

3 Learning Theory **19**
3.1 Strings and Languages . . . 20
3.2 Automata Learning . . . 21
3.3 L^* Algorithm . . . 21
3.4 *ID* Algorithm . . . 23
3.5 *IDS* Algorithm . . . 24
3.6 L^* *Mealy* Algorithm . . . 25
3.7 Other Algorithms . . . 26
3.8 Basic Complexity Results . . . 29

4 Model Checking **31**
4.1 Introduction . . . 31
4.2 Basic Ideas . . . 31

4.3 Temporal Logic 32
4.4 Model Checking 33
4.5 NuSMV Model Checker 34

5 Conclusions and Future Work **37**
5.1 Summary 37
5.2 Contributions of the thesis 38
5.3 Author's personal contribution 38
5.4 Future Work 38

Bibliography **41**

II Included Papers **47**

A Paper 1 (Correctness and Performance of an Incremental Learning Algorithm for Finite Automata) **49**

B Paper 2 (Incremental Learning-based Testing for Reactive Systems) **69**

Part I

Introduction and Literature Survey

Chapter 1

Introduction and Background to Software Testing

1.1 Introduction

The purpose of software testing is to show that a software program works as desired and to possibly detect any defects before its delivery. For testing purposes, a program is executed with artificial data commonly known as *test cases* to spot errors and identify anomalies. According to [Sommerville 2009] software testing has two distinct goals: 1) to show to the customer and developer that software meets its requirements and 2) to identify incorrect behaviour in the software with respect to a specification. The former is referred to as *validation testing* and the latter as *defect testing*.

Testing is a part of a broader paradigm of software verification and validation to ensure the quality of the software end product. The subtle difference between software verification and validation was described in [Boehm 1979]. According to this description, validation aims to get the answer to the question whether we are building the right product. Verification on the other hand refers to ascertaining whether the product being built is correct according to some requirement specification. Validation is a more generic term intended for customer satisfaction and verification is a more specific term intended to ensure the correct behaviour of the software system according to specifications. Since testing is seldom exhaustive we cannot conclusively claim the absence of bugs after testing a software product although it is a good approach to locate the bugs (see [Dijkstra et al. 1972]).

Testing can concern both functional and non-functional requirements of software and it can begin either during the software development process or after coding has been completed. When testing is done during the development process then it involves the use of static techniques like *reviews*, *walkthroughs* or *inspections*. When the software product is tested after the completion of coding then it usually involves the use of dynamic testing techniques. In these the behaviour of the software is

observed and compared with the requirements by actually running or executing the software.

In this thesis a new approach to *specification-based black-box* testing of systems is considered called *learning-based testing (LBT).* In LBT , we use a learning algorithm (described in Chapter 3) to iteratively learn the *system under test (SUT).* The iteratively learned model is then *model checked* (described in Chapter 4) against a specific requirement formula expressed in *temporal logic* (described in Section 4.3). Any violation to this formula is treated as a *test case* and applied to the SUT on the next iteration of learning. The *pass* or *fail* verdict of the test case is *automatically* decided by an *oracle* (described in Section 1.2) using requirement formula and the outcome of the test case. Before we consider LBT in more detail (in Section 1.6), it is appropriate to begin by reviewing different testing techniques.

1.2 Specification Based Testing

The development of any engineering system begins with a *specification* of what it is required to do. A specification is an agreement between the developers and other stakeholders (who want the system to be developed). The stakeholder's focus is on *what* the system should do and developers address the question *how* the system will be built. The term specification has a precise meaning in traditional engineering fields but its meaning may vary depending upon the context it is used in case of software engineering. For example it is common to hear terms like *requirements specification*, *design specification* and *module specification* etc in software engineering. All these terms are used during different phases of software development and have a different meaning depending upon their context.

A specification can be described in a *formal* or an *informal* way. In an informal way, the description is in natural language and can use visual aids like diagrams, tables and other visual notations to enhance their understanding. On the other hand a formal description of a specification requires a precise syntax and semantics that can adequately capture the functionality of the system to be developed.

In specification based testing a set of test cases is generated from the specification which are then executed on the *System Under Test (SUT)* and its output observed and compared with the specification. The *verdict* about the test being either *pass* or *fail* is given by an *oracle.* An oracle can either be *manual* or *automated*. In the case of a *manual oracle*, a human decides the pass or fail verdict for the test. A test is a pass if the observed value of the test is from the expected set of values given in the test case description, otherwise the verdict is a fail. A manual oracle can however be slow, time consuming, even error prone and sometimes impossible for exhaustive testing. A manual oracle is used in case of informal specifications because automating the testing process from them is not trivial . However it is possible to use an *automated oracle* in case of formal specifications, in this case an oracle is an implementation of some criteria that compares an observed value against the set of expected values. It gives a verdict as pass if the expected set of

values contains the observed values, otherwise the verdict is a fail.

1.3 Black Box and Glass Box Testing

Black Box Testing

The tester may have to use different sets of testing techniques depending upon the availability or non-availability of source code. When the software tester doesn't have access to the source code then the software is treated as a *black box* and testing techniques used in this case are called *black-box testing* or *functional testing*. In this case SUT functionality is tested against a set of requirements and behavioral errors corresponding to inputs are recorded. This kind of testing requires a test set either generated automatically or manually depending upon whether the requirements are formal or informal respectively. The verdict is given as pass or fail depending upon the observed values and the expected values given in the test set. Different types of *black box* testing include *equivalence partitioning, boundary value analysis*, *all pairs testing, model-based testing, exploratory testing* (see [Jorgensen 2007]), *specification-based testing* (see Section 1.2) and *random testing* (see Section 1.3).

Random Testing

Random testing is thought to be the opposite of *systematic* testing like black-box testing or white-box testing. This is because of the fact that the word *random* is associated with meanings of derogatory nature such as "having no specific pattern" or "without a governing purpose" and so on. But in practice its use is described in [Hamlet 2002] as, "*Random testing, of course, is the most used and least useful method.*"

The question however is why random technique should be used instead of systematic testing technique? In [Hamlet 2002] two reasons have been described for the usefulness of test case generation through a random approach. First, algorithms exist for the selection of random points through pseudo random numbers which are useful in defining a vast number of test cases. Secondly, the statistical independence among test points enables statistical prediction of observations upon them. The former can be compromised since the pass or fail of an easily generated test case may not be that easily computable by the oracle. The latter however is useful in the context of software testing theory. This is because in a physical context of measurement only random fluctuations can be averaged out and refined to yield a better result over several trials or experiments. This may not be the case of systematic fluctuations as a result of some systematic testing approach especially when their cause or (even existence) is not identifiable (because system is treated as a black box) which will render the measurement invalid forever. Therefore random testing can be used as an effective tool when large number of test cases have to be generated and for benchmarking of other testing techniques against random testing because of its better statistical background.

White Box Testing

When the tester has access to internal data structures, underlying algorithms and the source code that implement them then *white-box testing* techniques are used (also called *glass-box testing*).

The major advantage of *white-box* testing is the possibility to define SUT *coverage* by a set of test cases representing the test requirements. Elements of graph theory have been used quite efficiently for this purpose. To meet this end a graph model of the SUT is set up and then coverage of the system by test cases is described in terms of e.g *node coverage, edge coverage or edge-pair coverage* (see [Amman and Offutt 2008]). Node coverage means the ability of the test suite to cover all reachable nodes in the graph of SUT. Similarly edge coverage and edge-pair coverage mean that the test cases in a *test suite* should be able to contain each reachable path of length ≥ 1 and length ≥ 2 respectively in the underlying graph of SUT. The quality of a test suite can also be determined by the extent of functional coverage achieved i.e how many functions or statements it is able to execute and test successfully. The former is called *function coverage* and the latter is called *statement coverage*. The *completeness* of the test suite created with *black box* testing can also be checked with this criteria.

The behaviour of black-box testing and white-box testing can be contrasted in terms of *scalability* and *testing from requirements.* White-box testing techniques are very good when it comes to describe the coverage of an SUT achieved. But these are not particularly good when the test suite is to be scaled for large systems. Such systems can possibly consist of thousands of paths with hundreds of selection statements and loops. It is not possible to test all paths of loops in such programs which renders exhaustive testing of such systems impossible. Similarly white-box testing approach does not provide the possibility of generating the test suite from requirements as it is meant to test different paths of the system. On both these counts black-box testing fares much better than white-box testing.

1.4 Conformance Testing

Conformance testing is a success story in specification-based testing and is widely used in telecom industry. The aim of conformance testing is to check whether a given SUT *conforms* to a formal specification or not. This type of testing is quite common for protocol testing and protocols are quite similar to reactive systems. A framework for conformance testing of protocols is for example [Tretmans 1996]. The notion of *conformance* has to be defined formally in this context. A conformance relation will precisely describe under what conditions an SUT conforms to a specification. For example when the specification allows two possible outputs for a particular input then the corresponding *conformance relation* can be defined either as allowing only one output for that input or showing no output at all in the implementation. More precisely we can say that an implementation I conforms to a specification S when at any point during the execution it is able to handle

at least as many inputs as the specification and at most as many outputs as the specification.

A test suite is also generated from the specification. The behaviour of the *SUT* is observed by executing test cases on it from the test suite. The pass and fail verdict for a test case from the test suite is decided by a verdict function which formally models observations of test execution with a test execution procedure.

A test suite is *sound* if every implementation that conforms to the specification also passes the test suite. Conversely a test suite is *complete* if every non-conforming implementation fails the test suite. For practical reasons it is impossible to achieve *completeness* because every non-trivial system will require infinite number of test cases to be executed before reaching completeness. Therefore an incomplete test suite should at least be *consistent*. This means that it should pass all the correct implementations and fail implementations showing errant behaviour. A good automatic test case generation tool should be able to produce test suites that are sound from a given specification.

1.5 Model Based Testing

Model-based testing (MBT) (see [Utting and Legeard 2006]) involves the use of a *design model* to guide software testing by executing the necessary artifacts. The model for testing purposes is an abstraction of the SUT but it should essentially describe all aspects needed for testing i.e test cases and their execution environment. The test cases so derived from this abstract model are also abstract and are part of an *abstract test suite (ATS)*. The ATS cannot however be executed on an SUT directly rather it has to be converted into an *executable test suite (ETS)* by some means for execution on a concrete SUT.

Since test cases are derived from models in case of MBT and not from source code. Therefore, MBT is generally considered as a form of black box testing. Nevertheless this approach allows us to define model coverage measures (again by using e.g graph theory).

Model-based testing can be carried out either *online* or *offline*. When it is online then the model-based testing tool acts directly on an SUT and executes the test cases on it (conversion from abstract test cases to concrete ones is done automatically by the tool). In offline model-based testing on the other hand the testing tool will generate test cases without actually executing them on an SUT. The test suite can be generated at some point in time and can be deployed and executed on the SUT at a later time.

The MBT approach can be used efficiently for the purpose of test automation provided the model is a formal and adequate behavioral description of the SUT which is also machine readable. Then it is possible to extract test cases automatically. There are several algorithmic methods for the extraction of test suites from *formal* descriptions of models which include test case generation by theorem prov-

ing (see [Helke et al. 1997]), constraint logic programming and symbolic execution (see [Offut 1991]) and more recently model checking (see [Fraser et al. 2009]).

1.6 Learning Based Testing

Learning-based testing (LBT) which is the subject of this thesis is an iterative approach to automate specification-based black-box testing. The LBT framework consists of the following:

- an SUT which is a black box
- a formal specification for SUT
- a learned model M of SUT

The former two are common to all specification-based testing. The latter however is a distinctive feature of LBT only. The LBT approach is a *heuristic iterative* approach which is based on the concept of learning a black-box SUT using *tests* as *queries.*

An LBT algorithm will work by executing test case inputs on an SUT. Let us say after the execution of n test case inputs $i_1, ..., i_n$ on the SUT outputs $o_1, ..., o_n$ have been observed. The learning algorithm will synthesize these n input/output pairs into a learned model M_n of the SUT. The learned model M_n is then satisfiability checked against the formal specification for SUT and a counterexample is returned in case of a fail. The counterexample will become input i_{n+1} for the SUT and after its execution output o_{n+1} is observed. If the SUT fails this test case i.e (i_{n+1}, o_{n+1}) does not satisfy the formal specification then the LBT algorithm terminates with a *true negative.* If this test is a pass then M_n was an inaccurate model and the test case was a *false negative* and the LBT algorithm goes on to construct a refined model M_{n+1}.

The LBT paradigm has been applied to both procedural SUTs in [Meinke 2004] and [Meinke and Niu 2010] and to reactive SUTs in [Meinke and Sindhu 2011].

A combination of *learning* and *model checking* has been considered in several earlier works in the literature to test or formally verify *reactive systems* see e.g [Peled et al. 1999], [Groce et al. 2006] and [Raffelt et al. 2008]. Reactive systems, learning and model checking will be reviewed in Chapter 2, Chapter 3 and Chapter 4 of this thesis respectively. Learning and model checking is also used in an approach called *counterexample guided abstraction refinement (CEGAR)* for verification within the model checking community (this will be discussed in Section 1.9) . The LBT approach is distinct from the above mentioned in the sense that its focus is on testing rather than verification. Its effectiveness is also significantly enhanced by the use of *incremental learning algorithms* which form a central theme of this thesis.

1.7 Inductive Testing

Inductive testing is a heuristic approach to black-box testing. The heuristic idea of inductive testing is also to learn a black box system using tests as queries. However testing in this case is done without specifications. It is based on the idea that software testing and inductive inference are the opposite sides of the same coin. In software testing we try to find the optimum (finite) number of tests that are sufficient to test the whole system. In computational learning (which provides the algorithm for inductive testing) we try to find the minimum number of queries sufficient to learn the whole system. While *inductive inference* aims at finding the minimal behavioral representation of the system by executing a finite sample of examples for the system. The common feature of learning and testing is very aptly described in [Walkinshaw et al. 2010] as follows: "The success of techniques in either area depends on the depth and breadth of the set of examples or tests". The likelihood of finding a bug or an inferred model is greater if the range of tests or examples is broader.

Inductive testing therefore is based on the idea of constructing test cases through a learning procedure. An inferred model will represent what has already been tested and the test case generator will try to find new tests that learn unknown parts of the SUT. The process of inductive testing is often terminated by means of an equivalence test between the learned model and the SUT.

In [Walkinshaw et al. 2010] it has been shown that inductive testing can achieve better functional coverage than random testing techniques and that it can be applied to large systems. The applicability of this approach was demonstrated by generating a test set for the Linux TCP/IP stack.

1.8 Static Checking

We will contrast specification based testing with other methods for software quality assurance such as formal methods based testing techniques including static checking and verification. This is because: 1) they also use specifications and 2) they use similar algorithms to those used in testing such as model checking and constraint solving.

Static checking involves the use of program evaluation techniques without actually running the software. This contrasts with dynamic execution used in testing. It can be done manually or with a tool in which case it is called automated static checking. These techniques are not an alternative to testing but are complimentary to it. Testing aims to find bugs while static checking/verification is used to prove program correctness. Testing, static checking and scalability can also be contrasted in terms of scalability. Verification is effective and used on small scales only but the analysis reached is very strong as compared to static checking. Static checking can be effectively used on a larger scale but the analysis is weaker compared to full verification. Testing on the other hand can be used on a very large scale but the

analysis about the behaviour of the program is compromised compared to static checking and verification techniques.

Manual Static Checking

When done manually, approaches like *reviews*, *walkthroughs* and *inspections* are able to detect bugs in the software.

A review may be done by the programmer , in which (s)he analyzes the logic of the program by examining the source code. Such a review is called a *code review*. A code review not only helps to spot and fix potential mistakes in the software product, but it also improves the programmer's skills. Common code problems identified with this technique include race conditions, memory leaks, buffer overflows etc. A review may also be carried out in collaboration with a colleague in which case it is called a *peer review*. If the colleague happens to be more experienced than the programmer can get useful feedback not only in terms of finding potential bugs but also about optimizing sections of code by using more efficient programming constructs. *Pair programming* (two programmers code together), *over-the-shoulder* (one programmer looks over the shoulder of the author when he goes through the code) and *lightweight* code reviews are other examples of a review used in static checking.

A *walkthrough* is a kind of peer review different from the peer review discussed above in which a developer leads interested stakeholders through a software product and they ask questions and provide comments about possible problems. It differs from a review primarily because direct suggestions for improvement can be obtained, participants get familiar with the product and it omits product and process metrics.

An *inspection* on the other hand involves a peer review conducted by well trained individuals who analyze the software product for defects using a well defined process. Inspectors try to reach on a consensus on a work product like *software requirements specifications* and *test plans*. A defect in an inspection will be anything that will keep the inspector from approving that work product.

Automated Static Checking

Automated static checking or static program analysis is another technique used for the analysis of of computer programs without actually executing them. This is done with the help of a tool which analyzes the source code or the object code of a program to spot potential problems within the program. The sophistication of these tools vary from one to the other as some tools analyze individual statements and declarations, while other tools analyze complete source code. The information contained in this analysis varies from tool to tool. It may simply consist of highlighting code errors (e.g. as shown by the Lint tool which analyzes *C* code) to more complicated programs that prove properties of a program mathematically (e.g *MAPLAS* (see [Wichmann et al. 1995]) tool uses directed graphs and regular

algebra to prove that software being analyzed meets its mathematical specification). *ESC/Java* (see [Flanagan et al. 2002]) is another well known tool (ESC stands for *Extended Static Checker*). It tries to find common run-time errors in Java programs at compile time. It is based on a theorem prover analysis and a simplified semantic model of Java code. Extended static checking which in this case means to statically check the correctness of constraints of a given program for example an integer being greater than zero or lying between upper and lower bounds of an array.

1.9 Full Verification

The type of static analysis in which properties of a program are proven mathematically is called *formal verification*. Its use is becoming increasingly widespread in industry for the verification of properties of software used in safety-critical computer systems. An important technique in this regard is called *model checking*. It considers finite state systems or those which can be reduced to finite state by some kind of abstraction technique. The model checking algorithm then checks this (abstract) model against temporal logic requirements. If it finds an error it can either report a counterexample to that specification or simply notify the error otherwise it will report the specification to be valid for the model. A more detailed description of model checking is given in Chapter 4.

CEGAR: Inductive Verification

There is a branch of *formal verification* which is particularly relevant to this thesis. Formal verification can be used to mathematically prove the properties of programs with techniques like *model checking*. But the model checking approach has its limitations such as *state space explosion* that can occur if the components of the system being verified make transitions in parallel. The problems of state space explosion were a reduced in severity by the introduction of *binary decision diagrams (BDDs)* see e.g. [Burch et al. 1992]. This approach was used in a well known model checker developed around that time called NuSMV see [NuSMV 2.5.2]. But the state space explosion problem has not completely been resolved yet despite the success of symbolic techniques. Several reduction techniques have been introduced and studied in this regard but a more flexible technique for handling this problem has been *abstraction* which intuitively means simplifying details or removing components from the original design that are not relevant to the property being verified.

In the usual abstraction based approaches abstractions are often constructed manually. This process can be time consuming and also error prone. But these two drawbacks can be eliminated if a learning-based approach is used. In a learning-based approach as described in Section 1.6 an abstraction can be built automatically by using counterexamples to steer the process of learning. This combination of learning and model checking is very similar to our own but the emphasis in CEGAR is on full verification (or at least static checking) and complete learning rather than testing.

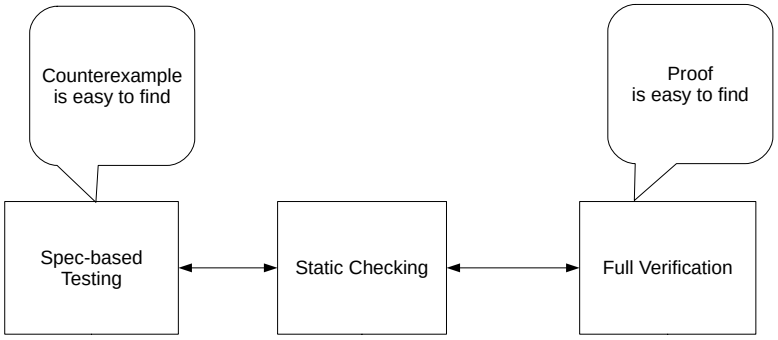

Figure 1.1: A comparison of Testing and Verification

A relative comparison of testing, static checking and full verification in terms of convenience in finding a counterexample or a full proof of correctness is given in the Figure 1.1. Moving left in the figure we approach spec-based testing where finding counterexamples is convenient rather than full proof of correctness. Moving right in the figure takes us closer to full verification where finding a proof of correctness is more convenient than to find a counterexample.

Chapter 2

Principles of Finite Automata

2.1 State Machines and Formal Languages

State machines can be used to describe the behaviour of a diverse class of computational systems e.g communication protocols, digital circuits, reactive systems and objects, and hence are of great significance. Therefore it will be useful to begin with a brief account of state machines. Let Σ be any set of symbols (aka alphabet) then Σ^* denotes the set of all finite strings over Σ including the empty string ϵ. The length of any string $\alpha \in \Sigma^*$ is denoted by $|\alpha|$ and $|\epsilon| = 0$. For any two strings $\alpha_1, \alpha_2 \in \Sigma^*$ $\alpha_1.\alpha_2$ denotes their concatenation.

Definition (Deterministic Finite Automata).

A deterministic finite automata *(DFA)* $\mathcal{A}$ is a quintuple $\langle Q, \Sigma, \delta, q_0, F \rangle$ where :

- Q is a finite set of states,
- Σ is a finite set of input symbols,
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $q_0 \in Q$ is the start state,
- $F \subseteq Q$ is the set of final states also called acceptor states.

Now δ can be inductively lifted to $\delta^* : Q \times \Sigma^* \to Q$, where $\delta(q, \epsilon) = q$ and $\delta^*(q, \sigma_1, ..., \sigma_n) = \delta(\delta^*(q, \sigma_1, ..., \sigma_{n-1}), \sigma_n)$. A string β of the form $\sigma_1, ..., \sigma_n$ is *accepted* by $\mathcal{A}$ iff $\delta^*(q_0, \beta) \in F$. The language accepted by $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, consists of all strings $\sigma_1 ... \sigma_n \in \Sigma^*$ which are accepted by $\mathcal{A}$ i.e $\delta^*(q_0, \sigma_1 ... \sigma_n) \in F$.

□

For any given DFA $\mathcal{A}$ there exists a minimum state DFA $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$ and is called a *canonical* DFA. It can be shown that a canonical DFA has one dead state at the most.

Several different generalizations of DFA have been proposed and studied to model different classes of systems. Important examples among such state machine models include *Moore machines*, *Mealy machines*, *Extended Finite State Machines (EFSM)* and *Kripke structures*. All these types of state machines can be designed to deal with either *deterministic* or *non-deterministic* behaviour depending upon the type of the system to be modelled.

In the case of Moore machines the output depends on the input only. While in the case of Mealy machines the output depends upon the input as well as the current state. A Kripke structure on the other hand is a specific type of state machine which uses a labelling function to label states corresponding to some atomic propositions i.e multi-bit output. In this thesis we will focus on Moore machines and Kripke structures. More precise definitions of both are given below.

Definition (Moore Machine).

A Moore machine M is a six-tuple such that $M = \langle Q, \Sigma, \Omega, q_0, \delta, \lambda \rangle$ where:

- Q is a finite set of states,
- $\Sigma = \{\sigma_1, ..., \sigma_n\}$ is a finite set of input symbols,
- $\Omega = \{\omega_1, ..., \omega_m\}$ is a finite set of output symbols,
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $\lambda : Q \to \Omega$ is an output function that maps states to the output symbols,
- $q_0 \in Q$ is the initial or start state.

Clearly δ can be inductively lifted to δ^* as in Definition 2.1.1.

□

Definition (Mealy Machine).

A Mealy machine $\mathcal{M}ly$ is a six-tuple $\mathcal{M}ly = \langle Q, \Sigma, \Omega, q_0, \delta, \lambda \rangle$ where:

- Q is a finite set of states,
- $\Sigma = \{\sigma_1, ..., \sigma_n\}$ is a finite set of input symbols,
- $\Omega = \{\omega_1, ..., \omega_m\}$ is a finite set of output symbols,
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $\lambda : Q \times \Sigma \to \Omega$ is the output function,

- $q_0 \in Q$ is the initial state.

Then Σ^* and Ω^* represent the set of all finite sequences of inputs and outputs over Σ and Ω respectively.

□

Definition (Kripke Structure).

A (non-deterministic) Kripke structure K over a set AP of atomic propositions is a five-tuple $K = \langle Q, \Sigma, \delta, q_0, \lambda \rangle$ where:

- Q is a finite set of states,
- $\Sigma = \{\sigma_1, ..., \sigma_n\}$ is a finite set of input symbols,
- $\delta \subseteq Q \times Q$ is a transition relation,
- $q_0 \in Q$ is the initial or start state,
- $\lambda : Q \to 2^{AP}$ is a labelling function for states.

We say that K is deterministic if δ is a function $\delta : Q \to Q$. Each property in AP describes some local property of system states $q \in Q$. Each state of the system is assigned a set of propositions by the labelling function λ.

□

Since we want to work with Kripke structures as Moore machines with states labelled by *Boolean* vectors the above definition of Kripke structures can be reformulated as follows.

Definition (Deterministic Kripke Structure).

A deterministic Kripke structure K is a five-tuple $\langle Q, \Sigma, \delta, q_0, \lambda \rangle$ where:

- Q is a finite set of states,
- $\Sigma = \{\sigma_1, ..., \sigma_n\}$ is a finite set of input symbols,
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $q_0 \in Q$ is the initial state,
- $\lambda : Q \to \mathbb{B}^k$ where $(b_1 ... b_k) \in \mathbb{B}^k$ is an enumeration or indexing of a set AP of k atomic propositions.

As in Definition 2.1 we let $\delta^* : Q \times \Sigma^* \to Q$ denote the iterated state transition function, where $\delta^*(q, \epsilon) = q$ and $\delta^*(q, \sigma_1, ..., \sigma_n) = \delta(\delta^*(q, \sigma_1, ..., \sigma_{n-1}), \sigma_n)$. Here we let $\lambda^* : \Sigma^* \to \mathbb{B}^k$ denote iterated output function $\lambda^*(\sigma_1, ..., \sigma_n) = \lambda(\delta^*(q_0, \sigma_1, ..., \sigma_n))$.

□

2.2 Reactive Systems

Reactive systems are systems which continuously interact with their environment, via a sequence of inputs and outputs. They typically execute cyclically in a loop, and during this process they take inputs from their environment and return values to the desired outputs according to the received inputs. Examples of such systems include embedded systems such as cruise controllers in vehicles, systems controlling mechanical devices like trains, air traffic control, medical devices or control system in a nuclear reactor etc. A graphical user interface for a computer program can also be thought as a type of reactive system. A distinctive feature of such systems is that their behaviour can be modeled as automata (like the algebraic structures discussed in Section 2.1). Also their behaviour can be specified in temporal logics (temporal logics will be reviewed in Section 4.3) such as linear temporal logic *(LTL)* and computational tree logic *(CTL)* etc. This makes them well suited to automatic verification through the use of model checkers or to testing with an automated testing technique. As examples of reactive systems we discuss two case studies which appear in paper 2, these are:

1. a simple cruise controller

2. a 3-floor elevator

A Cruise Controller

A cruise controller *(cc)* is an embedded safety critical software system which is commonly used in modern vehicles. A simplified model of such a cruise controller is given in Figure 2.1. The input set of the cruise controller *cc* consists of the set *{break, dec, gas, acc, button}*. The value *break* is used to reduce speed by the driver, *dec* and *acc* represent physical constraints on speed corresponding to external factors such as going uphill and downhill respectively, and *button* is used to turn on or turn off the *cc*. The *cc* has three modes namely manual, cruise and disengaged and these are represented by the first two bits of a bit vector consisting of 5-bits for this particular case. Therefore *mode=00* in the figure represents the manual mode, *mode=01* represents the cruise mode and *mode=10* represents the disengaged mode. Similarly, there are three strongly discretized values for speed which are *0,1* and *2* represented in a bit vector as *speed=00*, *speed=01* and *speed=10* respectively. This discretization of speed focuses on the essential switching properties of the cc. In manual mode the *cc* can take any of the allowed values of speed, in cruise mode only value *1* or *speed=01* is allowed and in disengaged mode the *cc* can have speed values of *0* or *speed=00* respresenting too low speed or *2* or *speed=10* representing too high speed. The last bit in the bit vector represents the *cc* button, which can have values *0* or *1* representing the *"off"* and *"on"* states of the button.

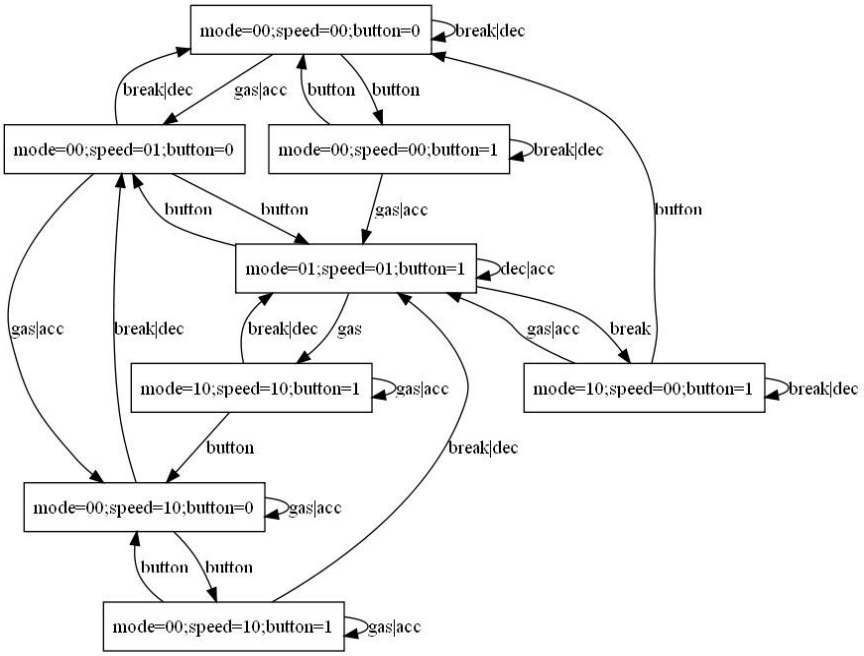

Figure 2.1: 5-bit Cruise Controller

A 3-Floor Elevator

The 3-Floor elevator model is another typical embedded safety critical system, this example has 38 states and an 8-bit vector for the output as shown in Figure 2.2. This particular representation is as a hierarchical statechart which can be flattened to a conventional FSM. The input alphabet set consists of the symbols *{c1, c2, c3, tick}* where *c1, c2* and *c3* represent the calls to first, second and third floors respectively and *tick* is a special input which models the clock representing the passage of time. The 8-bit output vector consists of bits denoted by *w1, w2, w3, cl, Stop, @1, @2,* and *@3* respectively, where *w1, w2* and *w3* represent queued calls to floors one, two and three respectively. The elevator door state is represented by *cl* and its negation *!cl* represents an open door. The elevator motion is represented by the *Boolean* variable *Stop* and its negation *!Stop* denotes the elevator is moving. Similarly the bits *@1, @2,* and *@3* represent the location of the elevator on floor 1, floor 2 and floor 3 respectively.

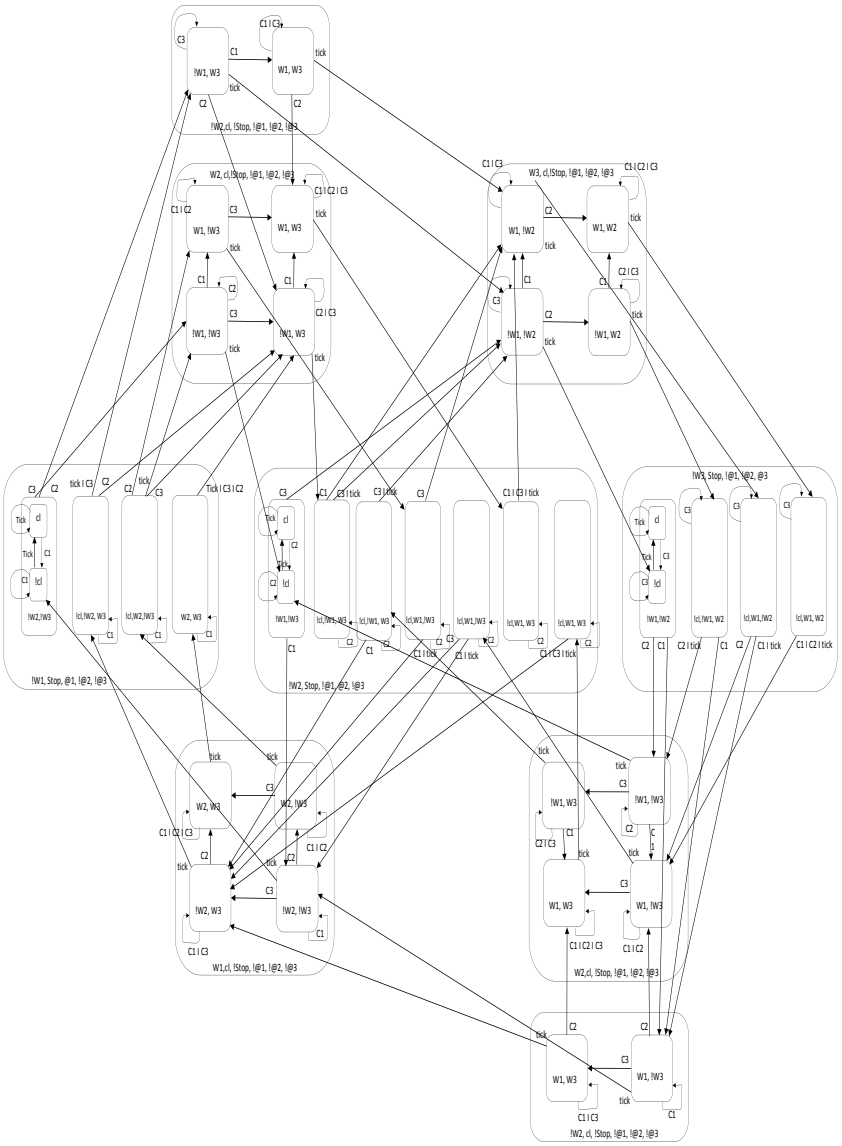

Figure 2.2: 3-Floor Elevator

Chapter 3

Learning Theory

Computational learning involves designing algorithms that attempt to infer a specific structure $s \in S$ (also called a target) from a set of structures S given some data $\mathcal{D} = d_1, ..., d_i, ..., d_n$ about s. The data is a set or sequence of data elements and a learning algorithm tries to either predict a response for some future unseen input or simply summarizes the behaviour corresponding to the seen input in a comprehensible manner as an *approximation* $h \in \mathcal{H}$ of the actual system. Both s and h are assumed to be functions taking input of the form $d \in D$. An approximation need not be *exact* and does not necessarily explain everything about the target. It is based on some subset of the inputs $d \in D$ on which both h and s closely agree (assuming approximation is not exact).

If $s \in S$ has known values for data elements $d_i \in D$ and these are used to actually construct *approximations* $h \in \mathcal{H}$ about $s \in S$ then such learning is called *supervised learning*. On the other hand if the function values $s(d_i)$ for $d_i \in D$ are not known then the data values are partitioned in an appropriate way by the learning algorithm and this type of learning is called *unsupervised learning*. The learning will be called *exact* if $s(d_i) = h(d_i)$ for all $d_i \in D$. *Exact* learning typically involves the use of an *adequate teacher* sometimes also called an *oracle* that can answer the queries $s(d_i) =?$ and whether $s(d_i) = h(d_i)$ for all $d_i \in D$. The former are called *membership queries* and the latter are called *equivalence queries*. Sometimes there can be separate oracles to answer both these types of queries. If an oracle answers equivalence queries then it may or may not provide counterexamples in case of a negative answer to an equivalence query. If a counterexample is provided by the oracle then the counterexample is from the set $s(d_i) \setminus h(d_i)$ or $h(d_i) \setminus s(d_i)$ for $d_i \in D$.

Learning can be either *active* or *passive* in the context of systems where the learner is interacting with its environment. Learning is termed active if the learning algorithm (learner) decides on which data points it has to receive a response from the environment. On the other hand if the environment provides responses to the learning algorithm (learner) on some data points without being asked by it then

the learning is termed passive.

A learning algorithm can be *complete* or *sequential* in nature. A complete learning algorithm will construct a single *hypothesis* h when it has gathered sufficient information about the target and asks the equivalence query $s = h$. A sequential learning algorithm on the other hand will construct a sequence of hypotheses $h_1, h_2...$ after each membership query which finitely converge to s i.e $h_n = s$ for some $n \in \mathbb{N}$. If each hypothesis h_{i+1} is able to use the information from hypothesis h_i then the learning algorithm is termed incremental. Therefore incremental learning is a special type of sequential learning. Equivalence queries in the case of incremental/sequential learning algorithms are used to terminate the algorithm when hypothesis h has become equal to the target s. The goal of complete learning is to *exactly* learn s while the goal of sequential learning is to make best guess at any time using available data.

Machine learning is used in several types of social, managerial and natural sciences such as artificial intelligence, pattern recognition, cognitive science, adaptive control and theoretical computer science to name a few. Many different computational structures can be learnt including *functions, logic programs* and *rule sets, finite state machines* and *grammars*. In this thesis we will focus on finite state machines and the algorithms that learn them. Some preliminaries concerning these are given in the following sections:-

3.1 Strings and Languages

Let Σ be a finite set of symbols then Σ^* denotes the set of all finite strings over Σ including the empty string ϵ. We let Σ^ω denote the set of all infinite strings $\sigma_0, \sigma_1, ...$. A string $\alpha \in \Sigma^*$, is termed a *prefix* of string γ if and only if there exists a string β such that $\gamma = \alpha\beta$ and we let $Pref(\gamma)$ denote the set of all prefixes of string γ. A subset of Σ^* is called a *formal language* and is denoted by $\mathcal{L}$. Recall from *Kleene's Theorem* that a formal language $\mathcal{L}$ accepted by a finite automaton *iff* $\mathcal{L}$ is a *regular language*. Let $\mathcal{L}_{reg1}$ and $\mathcal{L}_{reg2}$ vary over regular languages then a recursive definition of all regular languages can be given by:

$$\mathcal{L}_{reg} ::= \emptyset \mid \{\epsilon\} \mid \{a\} \mid \mathcal{L}_{reg1} \cup \mathcal{L}_{reg2} \mid \mathcal{L}_{reg1} \cdot \mathcal{L}_{reg2} \mid \mathcal{L}^*_{reg} \tag{3.1}$$

In other words, the empty language $\emptyset$, the empty string language ϵ , the singleton language $\{a\}$ where $a \in \Sigma^*$ are all regular languages. Similarly union "$\cup$" and concatenation "." of two regular languages $\mathcal{L}_{reg1}$ and $\mathcal{L}_{reg2}$ is also a regular language. The *Kleene star* construction $\mathcal{L}^*_{reg}$ applied to a regular language also gives a regular language.

If $\mathcal{S}_1$ and $\mathcal{S}_2$ are sets then $\mathcal{S}_1 \oplus \mathcal{S}_2$ denotes the symmetric difference of $\mathcal{S}_1$ and $\mathcal{S}_2$ which means those elements that are either in $\mathcal{S}_1$ or $\mathcal{S}_2$ but not both. The cardinality of the set $\mathcal{S}$ is denoted by $|\mathcal{S}|$.

3.2 Automata Learning

In this thesis our focus will be entirely on computational learning algorithms that infer state machines and Kripke structures. The inputs of such algorithms are strings that may be words of a regular language. For this reason these algorithms are also referred to as *regular inference* algorithms. In a typical regular inference algorithm there is a *Learner* which initially has no knowledge of the target $\mathcal{M}$. It starts by asking queries to a *Teacher* and an *Oracle*. There are two basic types of queries depending upon whether these are posed to a *Teacher* or an *Oracle*.

- A query to the Teacher is called a *membership query* in which the Learner asks whether a given string $\alpha \in \Sigma^*$ is in $\mathcal{L}(\mathcal{M})$.
- A query to the Oracle is called an *equivalence query* in which the Learner asks whether the approximation or hypothesis $\mathcal{H}$ has become equal to $\mathcal{M}$ or not. If they are not equal the Oracle will provide a counterexample either from $\mathcal{L}(\mathcal{M}) \setminus \mathcal{L}(\mathcal{H})$ or $\mathcal{L}(\mathcal{H}) \setminus \mathcal{L}(\mathcal{M})$.

There are many algorithms for learning state machines in the literature including [Gold 1967], [Trakhtenbrot and Barzdin 1973], [Angluin 1981], [Angluin 1987], [Rivest and Schapire 1993], [Dupont 1996], [Parekh et al. 1998], [Meinke 2010] and [Kearns and Vazirani 1994]. Most algorithms are for complete learning of *deterministic finite automaton DFA* such as [Angluin 1981] and [Angluin 1987]. However, some algorithms are for incremental learning such as [Parekh et al. 1998], [Dupont 1996] and some for sequential learning such as [Meinke 2010]. All these algorithms learn in the limit to yield a minimal approximation of the target. The concept of learning in the limit for DFA was first introduced by E. M Gold in [Gold 1967] where he showed that a regular language $\mathcal{L}(\mathcal{M})$ corresponding to a DFA $\mathcal{M}$ can be guessed by a finite number of wrong guesses(hypotheses) about $\mathcal{M}$ by using some inference or learning algorithm for DFA $\mathcal{M}$. This work led to several other contributions on the subject of learning theory and regular inference later on including the algorithms which we will discuss in the next sections.

In the following sections we will survey examples of learning algorithms which are particularly relevant to *learning based testing (LBT)*, these are *Angluin's* L^* algorithm introduced in [Angluin 1987] , *Angluin's ID* algorithm introduced in [Angluin 1981] and the *IDS* algorithm introduced in [Meinke and Sindhu 2010].

3.3 *L** Algorithm

Angluin's L^* algorithm see [Angluin 1987] is one of the classical complete learning algorithms in the literature on DFA learning. It accumulates information in the form of a finite collection of observations organized in an observation table $\mathcal{OT}$ which is a tuple $\mathcal{OT} = (P, S, T)$ for a given alphabet Σ such that:

- $P \subseteq \Sigma^*$ is a non-empty prefix closed set. A set is prefix closed if and only if every prefix of every member of the set is also a member of the set.

- $S \subseteq \Sigma^*$ is a non-empty suffix closed set. A set is suffix closed if and only if every suffix of every member of the set is also a member of the set.
- $T : ((P \cup P.\Sigma) \times S) \rightarrow \{acc, rej\}$ is a function which satisfies the property $ps = p's'$ implying $T(p,s) = T(p',s')$ for $p, p' \in P \cup P.\Sigma$ and $\forall s, s' \in S$.

The strings in $P \cup P.\Sigma$ are called *row labels* and strings in S are called *column labels*. The upper part of the observation table is indexed by P and the lower part is indexed by all strings which don't already appear in the upper part of the observation table and are of the form $p\alpha$ where $p \in P$ and $\alpha \in \Sigma$. The table is column-wise indexed by strings of a suffix-closed set S . Each row label $p \in P$ and each column label $s \in S$ is mapped to the set *{acc,rej}* by the function T. If $ps \in \mathcal{L}(\mathcal{M})$ then the entry field corresponding to that row label and column label will be *acc* otherwise it will be *rej*.

Function $row(p)$ is a finite function from S to $\{acc,rej\}$ for every $p \in (P \cup P.\Sigma)$ and is defined by $row(p)(s) = T(p,s)$ or more simply $row(p)$ represents the tuple of entries in the observation table corresponding to the row labelled p. All distinct rows of the form $row(p)$ where $p \in P$ represent the states of the hypothesis DFA. The hypothesis or approximation DFA can be constructed from the observation table using the rows labelled by $P.\Sigma$ to construct the transition function for the hypothesis DFA. Two conditions must however be fulfilled by the observation table $\mathcal{OT}$ for the successful construction of the hypothesis which are:

1. *closure*
2. *consistency*

An observation table $\mathcal{OT}$ is closed if for each $p_1 \in P.\Sigma$ there exists $p_2 \in P$ such that $row(p_1) = row(p_2)$ and $\mathcal{OT}$ is consistent provided that whenever $p_1, p_2 \in P$ such that $row(p_1) = row(p_2)$ then for all $\alpha \in \Sigma$, $row(p_1.\alpha) = row(p_2.\alpha)$. These are two examples of what we term *book keeping* queries, i.e queries generated internally by a learning algorithm. We may contrast these with queries generated externally by components such as:

- an equivalence oracle
- a data file
- a model checker
- a human being

When the observation table $\mathcal{OT}$ is closed and consistent then the hypothesis DFA $\mathcal{H}$ can be defined over alphabet Σ, with state set Q, initial state $q_0 \in Q$, accepting states $F \subseteq Q$ and the transition function δ by:

- $Q = \{row(p) : p \in P\}$,

- $q_0 = row(\epsilon)$
- $F = \{row(p) : p \in P \; and \; T(p) = acc\}$,
- $\delta(row(p), \alpha) = row(p.\alpha)$

The L^* algorithm maintains the observation table $\mathcal{OT}$ and the sets P and S are both initialized to $\{\epsilon\}$. Then L^* will perform membership queries for each $\alpha \in \Sigma$ and ϵ which will result in either an acc or rej for each query and corresponding fields in each row of $\mathcal{OT}$ are filled with these values. Afterwards $\mathcal{OT}$ is checked for consistency and closure. If it is not consistent then inconsistency is resolved by finding two strings $p_1, p_2 \in P$, $\alpha \in \Sigma$ and $s \in S$ such that $row(p_1) = row(p_2)$ but $T(p_1\alpha, s) \neq T(p_2\alpha, s)$ and setting the new suffix αs to S and filling in the missing fields by asking membership queries.

If $\mathcal{OT}$ is not closed then L^* finds $p \in P$ and $\alpha \in \Sigma$ such that $row(p\alpha) \neq row(p\prime)$ for all $p\prime \in P$ and appends $p\alpha$ to P. The missing fields in this case are also updated through membership queries.

After a number of membership queries, when $\mathcal{OT}$ has become consistent and closed then the hypothesis $\mathcal{H}$ can be constructed and checked for correctness against the target $\mathcal{M}$ by an equivalence query to the Oracle. If the answer to the equivalence query is "*yes*" then L^* terminates with a correct hypothesis $\mathcal{H}$ as output. Otherwise the Oracle will provide a counterexample β, such that $\beta \in \mathcal{L}(\mathcal{M}) \iff \beta \notin \mathcal{L}(\mathcal{H}))$ and L^* will extend $\mathcal{OT}$ with β and all its prefixes by asking membership queries.

3.4 *ID* Algorithm

The *ID* algorithm introduced in [Angluin 1981] is a complete learning algorithm. Unlike L^* it assumes the availability of a *live complete set* P of strings for the target *DFA* $\mathcal{A}$. A state $q_i \in Q$ is said to be *live* if there exists a string $\sigma_1, ..., \sigma_i \in \Sigma^*$ such that $\delta^*(q_0, \sigma_1, ..., \sigma_i) = q_i$ and $q_i \in F$. The string itself will be termed a *live* string and a set consisting of at least one such string for each live state of a given DFA is called a live complete set and is denoted by P. A state that is not live will be called a *dead state*. A canonical *DFA* has only one dead state.

The *ID* algorithm proceeds in the following steps:

1. Initializations: $i = 0$; $v_i = \epsilon$, $V = \{\epsilon\}$, $T = \{P \cup f((\alpha, \beta)|(\alpha, \beta) \in P \times \Sigma)\}$, where i is a counter that will count the number of distinguishing strings v, V is the set of all *distinguishing strings* v. P is the live complete set, f is a special concatenation function such that $f : P' \times \Sigma \to \Sigma'$, where $P' = P \cup \{d_0\}$ and $\Sigma' = \Sigma^* \cup \{d_0\}$. Therefore for any $\alpha \in \Sigma$ and $\beta \in \Sigma^*$ implies $f(d_0, \alpha) = d_0$ and $f(\alpha, \beta) = \alpha\beta$.

2. *ID* computes $T' = P' \cup \{f(\alpha, \beta)|(\alpha, \beta) \in P' \times \Sigma\}$, where $\alpha \in P'$ and $\beta \in \Sigma$ and $T = T' \setminus \{d_0\}$.

3. *ID* will construct a partition of set T' such that elements of T' that belong to the same state of $\mathcal{A}$ fall on the same block of partition of T' which is given by function E which is defined for i elements of set V such that $E_i = T' \rightarrow 2^V$ and $E_i(d_0) = \emptyset$ and $E_i(\alpha) = \{v_j | v_j \in V, 0 \leq j \leq i, \alpha v_j \in \mathcal{L}(\mathcal{A})\}$ for all $\alpha \in T'$.

4. Compute the function E_0 for $v_0 = \epsilon$, by setting $E(d_0) = \emptyset$ and for all $\alpha \in T$ if $\alpha \in \mathcal{L}(\mathcal{A})$ then set $E_0(\alpha) = \epsilon$ otherwise set it to $E_0(\alpha) = \emptyset$.

5. Once $E_i(\alpha)$ has been computed for all $\alpha \in T'$, *ID* searches for a pair $\alpha, \beta \in P'$ and a symbol $\sigma \in \Sigma$ such that $E_i(\alpha) = E_i(\beta)$ but $E_i(f(\alpha,\sigma)) \neq E_i(f(\beta,\sigma))$. If such a pair is found then $i+1th$ partition of T' is constructed by choosing some string $\gamma \in E_i(f(\alpha,\sigma)) \oplus E_i(f(\beta,\sigma))$ and a new distinguishing string $v_{i+1} = \sigma\gamma$ is defined by ID. The purpose of a distinguishing string is to identify states which have same behaviour for a particular string $\alpha \in \Sigma^*$ but have different behaviour for a suffix $\sigma \in \Sigma$. After identifying a distinguishing string $E_{i+1}(d_0)$ is set to $\emptyset$ and for each remaining $\alpha \in T'$, *ID* asks the query whether $\alpha v_{i+1} \in \mathcal{L}(\mathcal{A})$. If the answer is "yes" then $E_{i+1}(\alpha)$ is set to $E_i(\alpha) \cup \{v_{i+1}\}$ otherwise it is set to $E_i(\alpha)$.

6. If *ID* finds no such pair then $m = i$. Thus for all $\alpha, \beta \in P'$ and $\sigma \in \Sigma$, $E_m(\alpha) = E_m(\beta)$ implies $E_m(f(\alpha,\sigma)) = E_m(f(\beta,\sigma))$. *ID* then constructs the hypothesis DFA $\mathcal{H}$ which is isomorphic to $\mathcal{A}$ as under:

   a) states of $\mathcal{H}$ are all sets $E_m(\alpha)$ where $\alpha \in T'$

   b) the set $E_m(\epsilon)$ represents the initial state of $\mathcal{H}$

   c) The final states of $\mathcal{H}$ are the sets $E_m(\alpha)$ where $\alpha \in T$ and $\epsilon \in E_m(\alpha)$

   d) For all $\sigma \in \Sigma$ the transition relation δ of $\mathcal{H}$ is constructed by adding self loops to all states represented by $E_m(\alpha)$ if $E_m(\alpha) = \emptyset$ otherwise δ is set as $\delta(E_m(\alpha), \sigma) = E_m(f(\alpha,\sigma))$

7. ID outputs the description of hypothesis $\mathcal{H}$ and stops.

3.5 *IDS* Algorithm

The *IDS* algorithm introduced in [Meinke and Sindhu 2010] is an incremental learning algorithm . It takes its basic idea from the *ID* algorithm. But unlike the *ID* algorithm the *IDS* algorithm does not require the presence of a live complete set to start the inference procedure. It builds the hypothesis incrementally after each membership query. These membership queries are retrieved from a file S or an interface which gives a sequential order to the stream of examples. It assumes the availability of a learned teacher $\mathcal{A}$ (which is a DFA) like the previous two algorithms and it also assumes a stream of labelled examples as input. A labelled example for a DFA $\mathcal{A}$ is a pair such that $(\alpha, label(\alpha))$ where $\alpha \in \Sigma^*$ and $label(\alpha) = acc$ if $\alpha \in \mathcal{L}(\mathcal{A})$ and $label(\alpha) = rej$ if $\alpha \notin \mathcal{L}(\mathcal{A})$, the former is called a *positive* example for $\mathcal{A}$ and

later a *negative* example for $\mathcal{A}$. Thus *IDS* algorithm constructs a family of hypothesis $\mathcal{H}_1, \mathcal{H}_2, ...$ after reading each labelled example. Let $\mathcal{H}_m$ denotes the hypothesis inferred after observing m examples. Initially $\mathcal{H}_0$ is the initial hypothesis which is an automaton having transitions for all single character transitions in S, $\alpha \in \Sigma$ read from the initial state to corresponding next states. Afterwards $\mathcal{H}_0$ is extended for each labelled example $(\alpha, label(\alpha))$ received later. Each example is checked for consistency against $\mathcal{H}_m$ (i.e whether $\mathcal{H}_m$ correctly accepts/rejects α), if α is consistent with $\mathcal{H}_m$ then $\mathcal{H}_{m+1} = \mathcal{H}_m$ otherwise $\mathcal{H}_m$ is suitably modified to yield $\mathcal{H}_{m+1}$ which is consistent with α. The steps for *IDS* algorithm are given below the sets P, P', T, T', V and functions $Pref(\alpha)$, $E_i(\alpha)$, $f(\alpha, \beta)$ where $\alpha \in \Sigma^*$and $\beta \in \Sigma$ will remain the same as in Section 3.4:

1. Initializations: $i = 0$; $k = 0$; $m = 0$; $v_0 = \epsilon$, $V = \{v_0\}$
2. $P_0 = \{\epsilon\}$, $P'_0 = P_0 \cup \{d_0\}$, $T_0 = P_0 \cup \Sigma$, $T'_0 = T_0 \cup \{d_0\}$
3. Set $E_0(d_0) = \emptyset$ and for all $\alpha \in T_0$ if $\alpha \in \mathcal{L}(\mathcal{A})$ then $E_0(\alpha) = \epsilon$, otherwise $E_0(\alpha) = \emptyset$.
4. Refine the partition of set T'_m as described in point 5 section 3.4 above.
5. Construct the representation of hypothesis automata $\mathcal{H}_m$ as described in point 6 section 3.4 above
6. Wait for a new labelled example $(\alpha, label(\alpha))$ and check its consistency with $\mathcal{H}_m$, if it is consistent then $\mathcal{H}_{m+1} = \mathcal{H}_m$ and go to step 6. Otherwise $k = k + 1$, $m = m+1$, $P_k = Pref(\alpha) \cup P_{k-1}$, $P'_k = P_{k-1} \cup \{d_0\}$, $T_k = T_{k-1} \cup Pref(\alpha) \cup \{f(\alpha, \beta) | (\alpha, \beta) \in P_k \setminus P_{k-1} \times \Sigma\}$, $T'_k = T_{k-1} \cup \{d_0\}$ and for all $\alpha \in T_k \setminus T_{k-1}$ fill in the entries of $E_i(\alpha)$ using membership queries according to the function definition: $E_i(\alpha) = \{v_j | 0 \leq j \leq i, \alpha v_j \in \mathcal{L}(\mathcal{A})\}$, go to step 4.

3.6 *L* Mealy* Algorithm

Another complete learning algorithm was given by [Niese 2003] for the inference of Mealy machines. It is different from the above algorithms in the sense that it is for multi-bit output. The above mentioned algorithms are all DFA learning algorithms and deal with one bit output.

This algorithm works under the same assumptions as L^*. In particular it requires the availability of an adequate teacher, an oracle to answer equivalence queries and it asks the same type of membership queries like as L^*. It constructs a hypothesis when suitable information has been accumulated in the observation table $\mathcal{OT}$ to construct one. The contrasting feature of this algorithm with L^* however is that it looks at the output symbols produced by the SUT in response to input strings rather than their accepted/rejected status by SUT as in L^*.

In Niese's approach the SUT can be assumed to be modeled by a target Mealy machine which we refer to as $\mathcal{M}ly_T = \langle \Sigma, \Omega_T, Q_T, q_0^T, \delta_T, \lambda_T \rangle$. The SUT will not

provide accept/reject responses instead it will provide a response ω from a set Ω_T of output symbols. The observation table $\mathcal{OT}$ entries will consist of strings from Ω_T^* . The function T maps row and column labels to strings of output symbols and can be defined as $T : ((P \cup P.\Sigma) \times S) \rightarrow \Omega_T^*$. $T(p, s\sigma)$ is changed to ω by iterative output function $\lambda_T(\delta_T(q_0^T, ps), \sigma) = \omega$ where $p \in P$, $s\sigma \in S$, $\sigma \in \Sigma$ and $\omega \in \Omega_T^*$. The range of function $row(p)$ is changed from $row(p) : S \rightarrow \{acc, rej\}$ to cater for the Mealy style output as $row(p) : S \rightarrow \Omega_T^*$ and the finite function $row(p)$ is defined as $row(s)(e) = T(s, e)$. When the observation table has become closed and consistent after the accumulation of adequate information $\mathcal{M}ly_{\mathcal{H}}$ can be constructed by:

- $\Omega = \{T(p, \sigma) | p \in P, \sigma \in \Sigma\}$
- $Q = \{row(p) | p \in P\}$
- $q_0 = row(\epsilon)$
- $\delta(row(p), \sigma) = row(p\sigma)$
- $\lambda(row(p), \sigma) = T(p, \sigma)$

After the construction of the hypothesis an equivalence query is run on this hypothesis and in case of negative answer from the equivalence oracle, a counterexample which in this case is an input with different output for $\mathcal{M}ly_{\mathcal{H}}$ and $\mathcal{M}ly$ is returned otherwise, in case of a positive answer from the equivalence oracle the L^* Mealy algorithm will terminate.

3.7 Other Algorithms

In addition to the L^*, *ID*, *IDS* and L^* *Mealy* algorithms which were described in the preceding sections there are many other learning algorithms as mentioned in Section 3.2. They have different features of their own depending upon the number of bits in the output and type of system (e.g Moore, Mealy, Kripke etc) they learn. Most of them are complete learning algorithms. From the point of view of testing, a complete learning algorithm is less efficient than an incremental or sequential learning algorithm for the following reasons:

1. Real software systems can be too complex to be completely learned in a reasonable time.
2. Testing of functional requirements in software systems often correspond to the identified use case(s) and therefore testing a particular requirement does not require the whole system to be tested.
3. Membership queries can be expensive in terms of execution time of the learning algorithm as observed in [Bohlin and Johnson 2008].

4. Due to the high cost of membership queries, ideally each one of them should be derived from the behavioral analysis of the hypothesis automaton (as these can become interesting test cases), while queries for internal book keeping should be minimal.

For these reasons we will focus on algorithms which construct hypotheses either incrementally or sequentially. Two algorithms already seem particularly useful in this context. The first was given by Dupont (see [Dupont 1996]) and the second was given by Meinke (see [Meinke 2010]). Both these algorithms are sequential in nature.

RPNI2 Algorithm

Dupont's *RPNI*2 algorithm is based on the concept of positive and negative inference which itself is an extension of *RPNI* algorithm which was introduced by Oncina and Garcia in [Oncina and Garcia 1992]. The RPNI algorithm requires the positive and negative information to be given as a whole which renders it irrelevant for learning based testing when new learning data becomes available as it continues from where it finished and the whole process has to be restarted. The *RPNI*2 algorithm removes this discrepancy of the RPNI algorithm and modifies it for a sequential setting, where negative and positive information is served to it in a random order and one at a time. It performs a recursive depth first search with backtracking of a lexicographic state set. The state set of the hypothesis is represented by computing an equivalence relation on input strings. It computes a quotient automaton, performs consistency checking by parsing and then constructs a non-deterministic hypothesis automaton which is later transformed into a deterministic automaton as output. Dupont showed that both these algorithms converge at the same rate and proved that RPNI*2* will learn in the limit.

CGE Algorithm

The *CGE* algorithm is a sequential learning algorithm for Mealy automata. It uses techniques from term rewriting theory and universal algebra to represent and manipulate automata using finite congruence generator sets. This algorithm has been proved to correctly learn in the limit. The CGE algorithm also performs a recursive depth first search of lexicographic state set with backtracking. But unlike RPNI2 which learns Moore automata the CGE algorithm is for learning Mealy automata. It uses a purely symbolic approach in which congruence generator sets are represented as string re-write systems (SRS) which are then used to compute normal forms of states. It therefore doesn't require the construction of quotient automata and a consistency check through parsing, further the hypothesis is always maintained as a deterministic automaton in CGE. In contrast to all four algorithms discussed in the previous sections both RPNI2 and CGE algorithms don't require any internal book keeping queries to yield a hypothesis.

IKL Algorithm

The *IKL* algorithm is a multi-bit extension of the *IDS* algorithm. A detailed version of this algorithm appears in Appendix B of this thesis. Here we discuss only the salient features of this algorithm. The *IDS* algorithm learns deterministic finite automata DFA with one bit output and *IKL* extends this to learn deterministic Kripke structures with *k-bit* output using the ideas of *bit slicing* and *lazy partition refinement*. This is essential for practical testing of reactive systems as such systems are not limited to one bit output. The approach used in *IKL* is to bit slice the output of a *k-bit* Kripke structure $\mathcal{A}$ to k individual Kripke structures $\mathcal{A}_1, ..., \mathcal{A}_k$ with 1-bit output. These component Kripke structures can be learnt by any regular inference algorithm such as *IDS*. The inferred Kripke structures $\mathcal{B}_1, ..., \mathcal{B}_k$ are recombined using a subdirect product into a k-bit Kripke structure which is behaviourally equivalent to $\mathcal{A}$. The basic idea of *IKL* is to construct a family of k different equivalence relations $E^1_{i_1}, ..., E^k_{i_k}$ in parallel for the elements of set T' representing state names which is shared among all 1-bit (bit sliced) Kripke structures. For each equivalence relation $E^j_{i_j}$ a set of distinguishing strings V_j is iteratively generated and equivalence classes in $E^j_{i_j}$ are modified until a congruence is reached. The concept of lazy partition refinement here means to reuse each distinguishing string v wherever possible to refine any equivalence relation $E^j_{i_j}$ which is not yet a congruence, on the other hand if it is already a congruence then it is not refined further. This helps in minimizing the internal book keeping queries of the *IKL* algorithm which are not useful from the perspective of testing as these usually do not make interesting test cases.

A summary which compares the features of the algorithms discussed so far is given in Table 3.1.

No	Algorithm	Learned Automata	Learning Type	Book Keeping Queries	Bits in output	Consistency Check	Closure Check
1	L^*	Moore	Complete	Yes	1	Yes	Yes
2	*ID*	Moore	Complete	Yes	1	Yes	Yes
3	*IDS*	Moore	Incremental	Yes	1	Yes	Yes
4	L^*Mealy	Mealy	Complete	Yes	$k \geq 1$	Yes	Yes
5	*RPNI2*	Moore	Incremental	No	1	Yes	Yes
6	*CGE*	Mealy	Sequential	No	$k \geq 1$	No	No
7	*IKL*	Moore	Incremental	Yes	$k \geq 1$	Yes	Yes

Table 3.1: Learning algorithm comparison

3.8 Basic Complexity Results

Since we want to use an efficient learning algorithm for our testing framework it is appropriate to discuss the complexity properties of these algorithms. The complexity of a learning algorithm is usually described in terms of queries generated by it to construct a hypothesis. The basic time complexities in terms of queries generated by the algorithms discussed above is shown below in the Table 3.2. The number of states in the automaton is represented by N, the length of the longest counterexample in case of L^* and L^*Mealy algorithms is denoted by M. The size of the set of input alphabet for the automaton used in the table is represented by $|\Sigma|$. The size of a live complete set in case of *ID* algorithm is represented by $|P|$ and similarly $|P_k|$ represents the size of the live complete set in case of the *IDS* and *IKL* algorithms for some k examples of the target and l in case of the *IKL* algorithm is the size of the bit vector. In case of *RPNI2*, $S_p \subseteq \mathcal{L}(\mathcal{A})$ is a positive sample and $|S_p|$ represents its size similarly $S_n \subseteq \mathcal{L}'(\mathcal{A})$ represents a negative sample and $|S_n|$ represents its size. In case of *CGE*, n represents the longest acyclic path in the automaton $\mathcal{A}$.

No	Algorithm	Time Complexity $O(Queries)$
1	L^*	$O(\|\Sigma\|.N^2M)$
2	*ID*	$O(\|\Sigma\|.\|P\|.N)$
3	*IDS*	$O(\|\Sigma\|.\|P_k\|.N)$
4	*IKL*	$O(\|\Sigma\|.\|P_k\|.Nl)$
5	L^*Mealy	$O(max(N,\Sigma).\|\Sigma\|.NM)$
6	$RPNI2$	$O((\|S_p\|+\|S_n\|)\|S_p\|^2)$
7	CGE	$O(\|\Sigma\|^{2n})$

Table 3.2: Learning algorithm time complexities

In a testing context, each query should ideally be an interesting test case but in practical situations this is not possible due to the reasons described at the beginning of Section 3.7. From a testing perspective a learning algorithm will be termed *efficient* compared to another learning algorithm if it can generate a query which can yield an interesting test case earlier than the others. In other words how many unique test cases it can generate before complete learning the target compared to other learning algorithms.

Chapter 4

Model Checking

4.1 Introduction

Although we will make use of model checking as a black-box, it is useful to have some insight into what model checkers do and how they do it. This chapter will cover a brief review of preliminaries in the field of model checking.

4.2 Basic Ideas

During the last decade or so the paradigm of model checking has become a powerful approach to automatic verification of a diverse class of systems e.g communication protocols, digital circuits, reactive systems etc. A model checker is an algorithm for analyzing the satisfiability of a logical formula ϕ that describes the behaviour of the system respresented as some kind of automaton usually a Kripke structure also called the model of the system and denoted $\mathcal{M}$. Usually the formula ϕ is taken from a temporal logic such as *LTL, CTL or CTL**. We will explain more about these logics in Section 4.3. Temporal logic is useful in describing dependencies between actions or events where one action or event is supposed to occur before or after another action or event. If the specification ϕ satisfies the model $\mathcal{M}$ then the model checker will report the specification to be true. On the other hand if the specification ϕ is violated by the model $\mathcal{M}$ then the model checker will report an error and may or may not construct a witness (a counterexample such as a path or a state) to the violation of property. Model checkers that report witnesses or counterexamples to the violation of a specification are especially useful in the field of automated testing because the counterexample can be used as an interesting test case input to SUT.

4.3 Temporal Logic

Temporal logics constitute systems of rules and notations pertaining to the represention of propositions qualified in terms of time which enable us to reason about such propositions. These logics are specific kinds of modal logics and have special modal operators to deal with time. The most common type of temporal logic is called *Linear Temporal Logic (LTL)* which was introduced into computer science by Amir Pnueli in [Pnueli 1977]. Other categories of temporal logics include *Computational Tree Logic (CTL)* also called *Branching Time Logic* introduced by [Clarke and Emerson 1982], *CTL** given by [Emerson and Halpern 1982] , *Hennesy & Milner Logic (HML)* given by [Hennessy and Milner 1985] and Modal μ-calculus given by [Kozen 1983] to name a few.

Linear Temporal Logic (LTL)

Linear time temporal logic or linear temporal logic (LTL) is the most commonly used logic in model checking and it provides us with connectives with which we can refer to time in the future. Time can be extended infinitely into the future as a discrete sequence of states with the help of LTL. Any such particular sequence of states is called a path of the system. Since the future may not be deterministic, there can be several such future sequences of states and any one of them may be the actual path of the system.

Definition.

A path $\pi :=< q_0, q_1, ... >$ in a deterministic Kripke structure K corresponding to an infinite word $w = \sigma_{0,}\sigma_1 ... \in \Sigma^\omega$ is an infinite sequence such that $\forall i \geq 0 :\ q_{i+1} = \delta(q_i, \sigma_i)$ for K and q_0 is the initial state of K.

Syntax of LTL

Definition.

Propositional linear temporal logic has the following syntax given in Backus Naur Form (BNF)

$$\phi ::= \bot \mid \top \mid p \in AP \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \rightarrow \phi_2 \mid \bigcirc\phi \mid \Diamond\phi \mid \Box\phi \mid \phi_1 \cup \phi_2 \quad (4.1)$$

The symbols $\bot$,$\top$, $\neg$, $\wedge$, $\vee$ and $\rightarrow$ are the usual Boolean connectives which have the same meaning in LTL as in propositional logic. The operators $\bigcirc$, $\Diamond$, $\Box$ and $\cup$ are the temporal connectives. Here $\bigcirc$ means that ϕ is true in the next state, $\Diamond$ means that ϕ is true in some future state, $\Box$ means that ϕ is true in all future

states including the current state and $\cup$ is the binary LTL operator which means that ϕ_1 will remain true until the state where ϕ_2 becomes true is reached. We can also mention two more LTL operators W and R which stand for *Weak Until* and *Release* respectively however they will not be used in this thesis. The precise semantics of the LTL formulas above is given in next section.

Semantics of LTL

Let K be a deterministic Kripke structure which models our system and $\phi \in LTL$ be a property we want to investigate. If ϕ is satisfied by the path π in K we can write, $K, \pi \models \phi$ or simply $\pi \models \phi$ if K is obvious from the context. On the other hand we write $K, \pi \not\models \phi$ if ϕ is not satisfied by the path π in the model K or simply $\pi \not\models \phi$.

We let $Paths(K, q_0)$ denote the set of all paths in our model K starting in state q_0, where $q_0 \in Q$ is the initial state of K.

Definition.

For a given deterministic Kripke structure $K = \langle Q, \Sigma, \delta, q_0, \lambda \rangle$, and a given path $\pi \in Paths(K, q)$ where $p \in AP$ and $q \in Q$, the satisfaction relation $K, \pi \models \phi$ for LTL formulas is inductively defined on the structure of ϕ as follows:

$$K, \pi \not\models \bot \tag{4.2}$$
$$K, \pi \models \top \tag{4.3}$$
$$K, \pi \models p \iff p \in \delta^*(q_0, \sigma_0, ..., \sigma_i) \tag{4.4}$$
$$K, \pi \models \neg\phi \iff K, \pi \not\models \phi \tag{4.5}$$
$$K, \pi \models \phi_1 \wedge \phi_2 \iff K, \pi \models \phi_1 \wedge K, \pi \models \phi_2 \tag{4.6}$$
$$K, \pi \models \phi_1 \vee \phi_2 \iff K, \pi \models \phi_1 \vee K, \pi \models \phi_2 \tag{4.7}$$
$$K, \pi \models \phi_1 \rightarrow \phi_2 \iff K, \pi \not\models \phi_1 \vee K, \pi \models \phi_2 \tag{4.8}$$
$$K, \pi \models \bigcirc\phi \iff K, \pi^1 \models \phi \tag{4.9}$$
$$K, \pi \models \Diamond\phi \iff \exists i \in \mathbb{N} : \; K, \pi^i \models \phi \tag{4.10}$$
$$K, \pi \models \Box\phi \iff \forall i \in \mathbb{N} : \; K, \pi^i \models \phi \tag{4.11}$$

4.4 Model Checking

Model checking aims to determine the correctness of a given property for a given model. Several algorithms have been designed and studied for this purpose such as *explicit model checking* see [Lichtenstein and Pnueli 1985], *symbolic model checking* see [McMillan 1993] and *bounded model checking* see [Biere et al. 1999]. These algorithms can verify correctness properties expressed in several different kinds of temporal logics including LTL discussed in Section 4.3.

Explicit model checking was the first successful approach developed for model checking. In this case the state space is explicitly represented and its forward exploration is done to discover the violation of a property. For verification of LTL properties for example the negation of an LTL property ϕ is represented as a Buchi automaton. If the language intersection for the model and Buchi automaton is not empty then this represents a violation of the property ϕ. This counterexample will be a path from the initial state to a state violating the property. This approach has been used in the *SPIN* model checker (see [Holzman 1997]).

Symbolic model checking uses *binary decision diagrams (BDD)* see [McMillan 1993] to model states and function relations on these states. This gives the advantage of expressing larger state spaces using this approach as compared to the explicit model checking approach. But on the other hand the large number of BDD variables impedes performance and the ordering of BDD variables adversely impacts the overall size of the described models.

The *bounded model checking* approach solves the model checking problem as a *constraint solving problem (CSP).* This allows the use of *satisfiability solvers (SAT)* to construct counterexamples up to a certain upperbound. As long as the boundary is not very big this approach is very efficient. The NuSMV model checker see [NuSMV 2.5.2] described in the next section uses the latter two approaches. We chose NuSMV instead of the SPIN model checker for the implementation of our *incremental learning-based testing* framework due to the following reasons:

- NuSMV uses the BDD approach which allows the representation of models with larger state space as compared to explicit state approach used in SPIN.

- Extracting counterexamples from the SPIN model checker is not trivial. This is because SPIN just provides a linear trace of states while the IKL learning algorithm requires input in the form of a string of input symbols read to reach that state. This can be handled in SPIN by modifications in the code of the described model to output the input alphabet read from each state while the model checker traverses that state. But still it will require special filtering code to extract the counterexample from the SPIN output after the verification of property. This job is however much less cumbersome in the case of NuSMV as it provides both the state trace and the input symbol trace from initial state to the state violating the property. The counterexample can be filtered from the NuSMV output with much more ease compared to SPIN. We do the filtering of counterexample from NuSMV output by using Java language's string manipulation features.

4.5 NuSMV Model Checker

The NuSMV [NuSMV 2.5.2] model checker is a symbolic model checker developed as a joint project by several universities of USA and Europe. It supports BDD based model checking see [McMillan 1993] as well as propositional satisfiability

(SAT) based model checking see [Biere et al. 1999]. It supports the expression of specifications in both LTL and CTL for both BDD based and SAT based model checking. It supports the use of heuristics to control state space explosion and enhance performance. The input language of NuSMV is SMV which is used to provide description of models.

SMV Language

The SMV language is the input language of the NuSMV model checker. It provides constructs to efficiently describe models as finite state machines. A small example of a simple cruise controller model description in SMV language is given below in Figure 4.1. The constructs of the SMV language that will be used in this thesis are *MODULE, VAR, IVAR, ASSIGN, SPEC* and *LTLSPEC*. The *MODULE* construct is used to define a method in SMV language. In our case we will be defining the main method with the help of this construct. The *VAR* construct is used to describe the state variables in the model. The *IVAR* construct is used to express the input variables in model. The *ASSIGN* construct is used to define the transitions between different states of the finite state machine which in our case will be a deterministic Kripke structure.The *LTLSPEC* construct is used to write an LTL formula against which the behaviour of the model will be verified by NuSMV. Commenting a piece of text is done by a double dash "–".

Expressing LTL in NuSMV

NuSMV also provides operators to express LTL formulas using the LTLSPEC reserved word. The global operator $\Box$ in LTL is written as G , the eventually or future operator $\Diamond$ is written as F and next operator $\bigcirc$ is written as X in SMV language. Some examples of such properties for reactive systems described in Section 2.2 are given below:

1. G(mode = 01 & speed = 01 & input = dec -> X(speed = 01))
2. G(mode = 01 & speed = 01 & input = gas -> X(mode = 10))

These two are the safety properties for the cruise controller described in Section 2.2. The property 1 describes the speed maintenance by cruise controller when going uphill. Here *mode = 01* (first 2 bits of the bit vector) means the vehicle is in cruise mode and *speed = 01* (3rd and 4th bits of the bit vector) means that the vehicle is moving with the allowable cruise speed. The *input = dec* means that vehicle is being decelerated externally like going uphill. The right side of the implication shows that the cruise controller should maintain its cruise speed while going uphill. The second property is also a safety property which shows that vehicle is disengaged (*mode = 10*) when gas-pedal is pressed *(input = gas)* in the cruise mode *(mode = 01)*.

```
MODULE main()

VAR

   mode : {manual, cruise, disengage};
   button : {on, off};
   break_pedal : boolean;
   gas_pedal : {0,1,2}; --cruise mode will work only for values 1.

ASSIGN

   init(mode) := manual;
   init(break_pedal) := FALSE;
   init(gas_pedal) := 0;

NEXT(mode) := case

   mode = manual & (gas_pedal = 1 | gas_pedal = 2 ) & button = off : manual;
   mode = manual & (gas_pedal = 1 ) & (button = off & next(button = on)) : cruise;
   mode = cruise & !break_pedal & (gas_pedal = 1 ) & button = on : cruise;
   mode = cruise & (break pedal | gas_pedal = 0 | gas_pedal = 2) & button = on : disengage;
   mode = disengage & ((button = on & next(button = off)) | gas_pedal = 0) : manual;
   TRUE : mode;
   esac;

LTLSPEC --one property at a time after this reserve word e.g the progress property shown below
   G(mode = manual → F (mode = cruise)) | G(mode = cruise → F (mode = manual ))
```

Figure 4.1: SMV code for a simple cruise controller

Chapter 5

Conclusions and Future Work

5.1 Summary

In chapters 1-4 we have provided a literature survey where we have seen the existing testing techniques and presented their salient features. We compared and contrasted the features of different testing and verification techniques. Both of these are intended to yield a defect free software. But testing alone (no matter how exhaustive) can not be used as a guarantee for correct software. Verification on the other hand is not feasible for large systems used in practice. But as they have a common goal but complementary nature there is a need to develop an "intermediate" approach. This approach should exploit complementary nature of both testing and verification. From this point of view, verification techniques such as model checking can be combined with a model inference/regular inference algorithm to generate test cases. Therefore a review of regular inference algorithms in the literature with their pros and cons and complexity properties in the context of software testing were discussed in Chapter 3.

In this thesis we have introduced a novel approach which combines the features of both testing and verification to test systems. This was done by integrating a verification tool (NuSMV model checker) with an incremental regular inference algorithm (*IKL*) for multi-bit output to generate test cases to yield the LBT framework. The *IKL* algorithm is an extension of the *IDS* algorithm which is for one bit output of DFA. The *IDS* algorithm and the experimental results to determine its suitability for learning-based testing are presented in Paper 1 appended with this thesis. Our research shows that making use of incremental learning for software testing is more efficient than existing similar approaches that use complete learning for this purpose. These results and the whole learning-based testing architecture can be seen in Paper 2 appended with this thesis.

5.2 Contributions of the thesis

The contribution of this thesis to the field can be summarized in the following points:-

- We developed new incremental learning algorithms for DFA and Kripke structures.
- a black-box specification-based testing architecture for reactive systems (following the LBT paradigm).
- an implementation of this architecture
- evaluation results which support the thesis that the LBT is an effective testing methodology.

5.3 Author's personal contribution

The work on this thesis led to the following two papers:

Paper 1

An abridged version of this paper with IDS algorithm and without some proofs have been submitted to the Conference on Algorithmic Learning Theory (ALT 2011).

The *IDS* algorithm presented in this paper was developed jointly during discussions with my supervisor. I also worked on parts of the proof of correctness. I did all the implementation and performed the experimental evaluation of this algorithm.

Paper 2

K. Meinke and M. Sindhu, *Incremental Learning-based Testing for Reactive Systems,* pp 134-151 in M. Gogolla and B. Wolff (eds) Proc Fifth Intl. Conf. on Tests and Proofs (TAP 2011) LNCS 6706, Springer Verlag, 2011.

The *IKL* algorithm for Moore machines with multi-bit output and the corresponding *LBT* framework presented in this paper were jointly developed during discussions with my supervisor. I did all the implementation and experimental evaluation of this framework.

5.4 Future Work

In the short term we envisage to do several optimizations to the current *LBT* framework. These include optimizing the oracle. Currently the oracle is used only for model checker generated queries but in future we could extend it to give a verdict on random as well as book keeping queries. This seems possible for some LTL properties. The *IKL* algorithm can also be improved further by using

a minimization on the product automaton. This will enhance performance of the system when the product automaton generated is relatively large and can be highly non-minimal. We also plan to automate bit slicing of specification formulas as currently this is done manually.

In the long term this research can be extended to several different areas which include:

- graphical requirements languages that replace temporal logic
- hybrid / realtime automata
- more complicated case studies (MBAT project)
- abstraction to deal with SUT complexity i.e. non Boolean SUTs

Bibliography

[Amman and Offutt 2008] P. Amman, J. Offut, *Introduction to Software Testing,* 1st Ed, Cambridge University Press, 2008. ISBN 978-0-521-88038-1

[Angluin 1981] D. Angluin, *A note on the number of queries needed to identify regular languages*, Information and Control, 51:76-87, 1981

[Angluin 1987] D. Angluin, *Learning regular sets from queries and counterexamples*, Information and Computation, 75:87-106, 1987

[Biere et al. 1999] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. *Symbolic model checking without bdds.* In Tools and Algorithms for Construction and Analysis of Systems, In TACAS'99, March 1999.

[Boehm 1979] B. W. Boehm, Software Engineering; *'R & D Trends and defence needs.'* In Research Directions in Software Technology. Wegner, P.(ed.). Cambridge, Mass.:MIT Press. 1-9.

[Bohlin and Johnson 2008] T. Bohlin, B. Johnson, *Regular inference for communication protocol entities*, Tech Report 2008-024, Dept. of Information Technology, Uppsala University, 2008.

[Burch et al. 1992] J. Burch, E. Clarke, K. McMillan, *Symbolic model checking: 1020 states and beyond.* Inf. Comput. 98, 142–170.

[Clarke et al. 2003] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, *Counterexample-guided Abstraction Refinement for Symbolic Model Checking.* Journal of the ACM, Vol 50, 2003.doi: 10.1145/876638.876643.

[Clarke and Emerson 1982] E M. Clarke and E. A Emerson. *Design and synthesis of synchronization skeletons using branching-time temporal logic.* In Logic of Programs, Workshop, pages 52-71, London, UK, 1982. Springer-Verlag. ISBN 3-540-11212-X

[DeMillo et al. 1978] R. A DeMillo, R. J Lipton and F. G Sayward, *Hints on test data selection: Help for the practicing programmer,* IEEE Computer, vol. 11, pp 34-41, April 1978.

[Dijkstra et al. 1972] E. W Dijkstra, O. J Dahl, C. A. R Hoare, *Structured Programming.* London: Academic Press 1972.

[Dupont 1996] P. Dupont, *Incremental regular inference*, pp 222-237 in: L.Miclet and C.Huguera (eds) Proceedings of the Third ICGI-96, LNAI 1147, Springer, 1996.

[Emerson and Halpern 1982] E. A Emerson and J. Y Halpern, *Decision procedures and expressiveness in the temporal logic branching time.* In STOC'82:Proceedings of the fourteenth annual ACM symposium on Theory of computing, pages 169-180, New York,NY,USA, 1982. ACM Press. ISBN 0-89791-070-2. doi:10.1145/800070.802190

[Flanagan et al. 2002] C. Flanagan, K.R.M Leino, M. Lillibridge, G. Nelson, J. B Saxe and R. Stata, *Extended static checking for Java*, In proc. Conference on Programming Language Design and Implementation, pages 234–245, 2002. doi: http://doi.acm.org/10.1145/512529.512558

[Fraser et al. 2009] G. Fraser, F. Watowa, P. E Ammann, *Testing with model checkers: a survey.* Software Testing, Verification and Reliability, 19(3): 215-261, 2009.

[Gold 1967] E. M. Gold, *Language identification in the limit.* Information and Control, 10(5):447-474,1967.

[Groce et al. 2006] A. Groce, D. Peled, M. Yannakakis: *Adaptive Model Checking.* Logic Journal of the IGPL 14(5): 729-744, 2006.

[Hamlet 2002] Hamlet. R, *Random Testing.* Encyclopedia of Software Engineering. 2002.

[Helke et al. 1997] S. Helke, T. Neustupny and T. Santen, *Automating Test Case Generation from Z Specifications with Isabelle, pp 52-71 in* LNCS, Springer-Verlag 1997.

[Hennessy and Milner 1985] M. Hennessy and R. Milner, *Algebraic laws for nondeterminism and concurrency.* J. ACM, 32(1):137-161, 1985. ISSN 0004-5411. doi: 10.1145/2455.2460

[Holzman 1997] G. J Holzman, *The model checker SPIN.* IEEE Trans. Softw. Eng., 23(5):279–295, 1997. ISSN 0098-5589. doi: 10.1109/32.588521.

[Jorgensen 2007] P. C Jorgensen, *Software Testing, A Craftman's Approach,* 3rd Ed, Auerbach Publications, 2007. ISBN 978-0-8493-7475-3

[Kearns and Vazirani 1994] M. J. Kearns and U. V. Vazirani, *An introduction to computational learning theory.* MIT Press, 1994.

[Kozen 1983] D. Kozen, *Results on the propositional mu-calculus.* Theor. Computational Science. 27:333-354, 1983

[Lichtenstein and Pnueli 1985] O. Lichtenstein and A. Pnueli, *Checking that finite state concurrent programs satisfy their linear specification. In POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107, New York, NY, USA, 1985. ACM Press. ISBN 0-89791-147-4. doi: 10.1145/318593.318622.

[McMillan 1993] K.L. McMillan. *Symbolic model checking.* In Kluwer Academic Publ., 1993.

[Meinke 2004] K. Meinke, *Automated Black-Box Testing of Functional Correctness using Function Approximation*, pp 143-153 in: G. Rothermel (ed) Proc. *ACM SIGSOFT Int. Symp. on Software Testing and Analysis,* ISSTA 2004, Software Engineering Notes 29(4), ACM Press, 2004.

[Meinke 2010] K. Meinke, *CGE: A Sequential Learning Algorithm for Mealy Automata* pp 148-162 in J. Sempere and P. Garcia (eds) Proc. Tenth Int. Colloq. on Grammatical Inference (ICGI 2010), LNCS, Springer Verlag, Berlin, 2010.

[Meinke and Niu 2010] K. Meinke and F. Niu, *A Learning-based approach to Unit Testing of Numerical Software*, in Proc. ICSTSS 2010, Lecture Notes in Computer Science, Springer Verlag, 2010.

[Meinke and Sindhu 2010] K. Meinke and M. Sindhu, *Correctness and Performance of an Incremental Learning Algorithm for Finite Automata*, technical report, School of Computer Science and Communication, Royal Institute of Technology, Stockholm, 2010.

[Meinke and Sindhu 2011] K. Meinke and M. Sindhu, *Incremental Learning-based Testing for Reactive Systems,* pp 134-151 in M. Gogolla and B. Wolf (eds) Proc Fifth Int. Conf on Tests and Proofs (TAP 2011) LNCS 6706, Springer Verlag, 2011.

[Niese 2003] Oliver Niese, *An integrated approach to testing complex systems.* Technical report, Dortmund University, 2003. Dissertation.

[NuSMV 2.5.2] The NuSMV Model Checker, http://nusmv.fbk.eu/

[Offut 1991] J. Offut, *Constraint-Based Automatic Test Data Generation.* IEEE Transactions on Software Engineering, 17:900-910, 1991.

[Oncina and Garcia 1992] J. Oncina, P. Garcia, *Inferring regular languages in polynomial update time.* In N. Perez de la Blanca, A. Sanfeliu, and E. Vidal, editors, Pattern Recognition and Image Analysis, volume 1 of Series in Machine Perception and Artificial Intelligence, pages 49-61. Word Scientific, 1992.

[Parekh et al. 1998] R.G. Parekh, C. Nichitiu and V. G Honavar. *A polynomial time incremental algorithm for regular grammar inference*, in: Proc. Fourth ICGI-98, Springer, 1998.

[Peled et al. 1999] D. Peled, M.Y. Vardi, M. Yannakakis, *Black-box Checking*, in J. Wu et al. (eds), *Formal Methods for Protocol Engineering and Distributed Systems,* FORTE/PSTV, 225-240, Beijing, 1999, Kluwer.

[Pnueli 1977] A. Pnueli, *The temporal logic of programs.* In 18th Annual Symposium on Foundations of Computer Science, 31 October-2 November, Providence, Rhode Island, USA, pages 46-56. IEEE, 1977

[Raffelt et al. 2008] H. Raffelt , B. Steffen and T. Margaria, *Dynamic Testing Via Automata Learning*, pp 136-152 in: Hardware and Software: Verification and Testing, Lecture Notes in Computer Science, Vol. 4899, Springer, 2008.

[Rivest and Schapire 1993] Ronald. L. Rivest and R. E. Schapire, *Inference of finite automata using homing sequences.* Information and Computation, 103:299-347, 1993.

[Sommerville 2009] I. Sommerville, *Software Engineering 9*, Pearson Edu. Addison-Wesley Publishers, 2009, Boston, , 9th Edition, ISBN 13:978-0-13-705346-9.

[Trakhtenbrot and Barzdin 1973] B. A. Trakhtenbrot and J. M. Barzdin, *Finite automata: behaviour and synthesis.* North-Holland, 1973.

[Tretmans 1996] J. Tretmans, *Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation.* Computer Networks and ISDN System, 29(1): 49-79, 1996. doi: 10.1016/S0169-7552(96)00017-7.

[Utting and Legeard 2006] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach,* Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2006. ISBN 0123725011

[Walkinshaw et al. 2010] N. Walkinshaw,K. Bogdanov, J. Derrick & J. Paris, *Increasing functional coverage by inductive testing: a case study* Proc. Twenty Second IFIP Int. Conf. on Testing Software and Systems (ICTSS 2010), Springer, 2010, 126-141

[Wichmann et al. 1995] B. Wichmann, A. Canning, D. L Clutterbuck, L. A Winsborrow, N. J Ward, D. W. R Marsh, *Industrial Perspective on Static Analysis,* Software Engineering Journal, March 1995.

Part II

Included Papers

Appendix A

Paper 1 (Correctness and Performance of an Incremental Learning Algorithm for Finite Automata)

Correctness and Performance of an Incremental Learning Algorithm for Finite Automata

Karl Meinke, Muddassar Azam Sindhu

School of Computer Science and Communication,
Royal Institute of Technology, 100-44 Stockholm, Sweden,
karlm@nada.kth.se, sindhu@csc.kth.se

Abstract. We present a new algorithm *IDS* for incremental learning of deterministic finite automata (DFA). This algorithm is based on the concept of distinguishing sequences introduced in [Angluin 1981]. We give a rigorous proof that two versions of this learning algorithm correctly learn in the limit. Finally we present an empirical performance analysis that compares these two algorithms, focussing on learning times and different types of learning queries. We conclude that *IDS* is an efficient algorithm for software engineering applications of automata learning, such as testing and model inference.

1 Introduction

In recent years, automata learning algorithms (aka. regular inference algorithms) have found new applications in software engineering such as *formal verification* (e.g. [Peled et al. 1999], [Clarke et. al. 2002], [Luecker 2006]) *software testing* (e.g. [Raffelt et al. 2008], [Meinke and Sindhu 2011]) and *model inference* (e.g. [Bohlin and Jonsson 2008]). These applications mostly centre around learning an abstraction of a complex software system which can then be statically analysed (e.g. by model checking) to determine behavioural correctness. Many of these applications can be improved by the use of learning procedures that are *incremental*.

An automata learning algorithm is incremental if: (i) it constructs a sequence of hypothesis automata $H_0, H_1, \ldots$ from a sequence of observations $o_0, o_1, \ldots$ about an unknown target automaton A, and this sequence of hypothesis automata finitely converges to A; and (ii) the construction of hypothesis H_i can reuse aspects of the construction of the previous hypothesis H_{i-1} (such as an equivalence relation on states). The notion of convergence in the limit, as a model of correct incremental learning originates in [Gold 1967].

Generally speaking, much of the literature on automata learning has focussed on *offline* learning from a fixed pre-existing data set describing the target automaton. Other approaches, such as [Angluin 1981] and [Angluin 1987] have considered *online* learning, where the data set can be extended by constructing and posing new queries. However, little attention has been paid to incremental

learning algorithms, which can be seen as a subclass of online algorithms where serial hypothesis construction using a sequence of increasing data sets is emphasized. The much smaller collection of known incremental algorithms includes the RPNI2 algorithm of [Dupont 1996], the IID algorithm of [Parekh et al. 1998] and the algorithm of [Porat, Feldman 1991]. However, the motivation for incremental learning from a software engineering perspective is strong, and can be summarised as follows:

(1) to analyse a large software system it may not be feasible (or even necessary) to learn the entire automaton model, and

(2) the choice of each relevant observation o_i about a large unknown software system often needs to be iteratively guided by analysis of the previous hypothesis model H_{i-1} for efficiency reasons.

Our research into efficient *learning-based testing* (LBT) for software systems (see e.g. [Meinke 2004], [Meinke, Niu 2010], [Meinke and Sindhu 2011]) has led us to investigate the use of *distinguishing sequences* to design incremental learning algorithms for DFA. Distinguishing sequences offer a rather minimal and flexible way to construct a state space partition, and hence a quotient automaton that represents a hypothesis H about the target DFA to be learned. Distinguishing sequences were first applied to derive the ID online learning algorithm for DFA in [Angluin 1981].

In this paper, we present a new algorithm *incremental distinguishing sequences* (*IDS*), which uses the distinguishing sequence technique for incremental learning of DFA. In [Meinke and Sindhu 2011] this algorithm has been successfully applied to learning based testing of reactive systems with demonstrated error discovery rates up to 4000 times faster than using non-incremental learning. Since little seems to have been published about the empirical performance of incremental learning algorithms, we consider this question too.

The structure of the paper is as follows. In Section 2, we review some essential mathematical preliminaries, including a presentation of Angluin's original ID algorithm, which is necessary to understand the correctness proof for *IDS*. In Section 3, we present two different versions of the *IDS* algorithm and prove their correctness. These are called: (1) *prefix free IDS*, and (2) *prefix closed IDS*. In Section 4, we compare the empirical performance of our two *IDS* algorithms with each other. Finally, in Section 5, we present some conclusions and discuss future directions for research.

1.1 Related Work

Distinguishing sequences were first applied to derive the ID online learning algorithm for DFA in [Angluin 1981]. The ID algorithm is not incremental, since only a single hypothesis automaton is ever produced. Later an incremental version IID of this algorithm was presented in [Parekh et al. 1998]. Like the IID algorithm, our *IDS* algorithm is incremental. However in contrast with IID, the *IDS* algorithm, and its proof of correctness are much simpler, and some technical errors in [Parekh et al. 1998] are also overcome.

Distinguishing sequences can be contrasted with the complete consistent table approach to partition construction as represented by the well known online learning algorithm L* of [Angluin 1987]. Unlike L*, distinguishing sequences dispose of the need for an equivalence oracle during learning. Instead, we can assume that the observation set P contains a *live complete set* of input strings (see Section 2.2 below for a technical definition). Furthermore, unlike L* distinguishing sequences do not require a complete table of queries before building the partition relation. In the context of software testing, both of these differences result in a much more efficient learning algorithm. In particular there is greater scope for using online queries that have been generated by other means (such as model checking). Moreover, since LBT is a black-box approach to software testing, then the use of an equivalence oracle contradicts the black-box methodology.

In [Dupont 1996], an incremental version RPNI2 of the RPNI offline learning algorithm of [Oncina and Garcia 1992] and [Lang 1992] is presented. The RPNI2 algorithm is much more complex than *IDS*. It includes a recursive depth first search of a lexicographically ordered state set with backtracking, and computation of a non-deterministic hypothesis automaton that is subsequently rendered deterministic. These operations have no counterpart in *IDS*. Thus *IDS* is easier to verify and can be quickly and easily implemented in practise.

The incremental learning algorithm introduced in [Porat, Feldman 1991] requires a lexicographic ordering on the presentation of online queries, which is less flexible than *IDS*, and indeed inappropriate for software engineering applications.

2 Preliminaries

2.1 Notation and Concepts for DFA

Let Σ be any set of symbols then Σ^* denotes the set of all finite strings over Σ including the empty string λ. The length of a string $\alpha \in \Sigma^*$ is denoted by $|\alpha|$ and $|\lambda| = 0$. For strings $\alpha, \beta \in \Sigma^*$, $\alpha\beta$ denotes their concatenation.

For $\alpha, \beta, \gamma \in \Sigma^*$, if $\alpha = \beta\gamma$ then β is termed a *prefix* of α and γ is termed a *suffix* of α. We let $Pref(\alpha)$ denote the prefix closure of α, i.e. the set of all prefixes of α. We can also apply prefix closure pointwise to any set of strings. The *set difference operation* between two sets U, V, denoted by $U - V$, is the set of all elements of U which are not members of V. The *symmetric difference* operation on pairs of sets is defined by $U \oplus V = (U - V) \cup (V - U)$.

A *deterministic finite automaton* (DFA) is a quintuple $A =< \Sigma, Q, F, q_0, \delta >$ where: Σ is the input alphabet, Q is the state set, $F \subseteq Q$ is the accepting state set and $q_0 \in Q$ is the starting state. The state transition function δ of A is a mapping $\delta : Q \times \Sigma \to Q$, and $\delta(q_i, b) = q_j$ means that when in state q_i given input b the automaton A will move to state q_j in one step. We extend the function δ to a mapping $\delta^* : Q \times \Sigma^* \to Q$ defined inductively by $\delta^*(q,\ \lambda) = q$ and $\delta^*(q,\ b_1,\ \ldots b_{n+1}) = \delta(\delta^*(q,\ b_1,\ \ldots b_n),\ b_{n+1})$. The *language* $L(A)$ *accepted by* A is the set of all strings $\alpha \in \Sigma^*$ such that $\delta^*(q_0,\ \alpha) \in F$. As is well known,

a language $L \subseteq \Sigma^*$ is accepted by a DFA if and only if, L is *regular*, i.e. L can be defined by a regular grammar. A state $q \in Q$ is said to be *live* if for some string $\alpha \in \Sigma^*$, $\delta^*(q,\ \alpha) \in F$, otherwise q is said to be *dead*. Given a distinguished *dead state* d_0 we define *string concatenation modulo the dead state* d_0, $f : \Sigma^* \cup \{d_0\} \times \Sigma \to \Sigma^* \cup \{d_0\}$, by $f(d_0,\ \sigma) = d_0$ and $f(\alpha,\ \sigma) = \alpha\ .\ \sigma$ for $\alpha \in \Sigma^*$. This function is used for automaton learning in Section 3. Given any DFA A there exists a unique minimum state DFA A' such that $L(A) = L(A')$ and this automaton is termed the *canonical DFA* for $L(A)$. A canonical DFA has at most one dead state.

2.2 The ID Algorithm

Our *IDS* algorithm is an incremental version of the ID learning algorithm for DFA introduced in [Angluin 1981]. The ID algorithm is an online learning algorithm for DFA that starts from a given live complete set $P \subseteq \Sigma^*$ of queries about the target automaton, and generates new queries until a state space partition can be constructed. Since the algorithmic ideas and proof of correctness of *IDS* are based upon those of ID itself, it is useful to review the ID algorithm here. Algorithm 1 presents the ID algorithm. Since this algorithm has been discussed at length in [Angluin 1981], our own presentation can be brief. A detailed proof of the correctness of ID and an analysis of its complexity can be found in [Angluin 1981].

A finite set $P \subseteq \Sigma^*$ of input strings is said to be *live complete* for a DFA A if for every live state $q \in Q$ there exists a string $\alpha \in P$ such that $\delta^*(q_0,\ \alpha) = q$. Given a live complete set P for a target automaton A, the essential idea of the ID algorithm is to first construct the set $T' = P \cup \{f(\alpha, b) | (\alpha, b) \in P \times \Sigma\} \cup \{d_0\}$ of all one element extensions of strings in P as a set of state names for the hypothesis automaton. The symbol d_0 is added as a name for the canonical dead state. This set of state names is then iteratively partitioned into sets $E_i(\alpha) \subseteq T'$ for $i = 0, 1, \ldots$ such that elements α, β of T' that denote the same state in A will occur in the same partition set, i.e. $E_i(\alpha) = E_i(\beta)$. This partition refinement can be proven to terminate and the resulting collection of sets forms a congruence on T'. Finally the ID algorithm constructs the hypothesis automaton as the resulting quotient automaton. The method used to refine the partition set is to iteratively construct a set V of *distinguishing strings*, such that no two distinct states of A have the same behaviour on all of V.

We will present the ID and *IDS* algorithms so that similar variables share the same names. This pedagogic device emphasises similarity in the behaviour of both algorithms. However, there are also important differences in behaviour. Thus, when analysing the behavioural properties of program variables we will carefully distinguish their context as e.g. v_n^{ID}, $E_n^{ID}(\alpha), \ldots$, and v_n^{IDS}, $E_n^{IDS}(\alpha), \ldots$ etc. Our proof of correctness for *IDS* will show how the learning behaviour of *IDS* on a sequence of input strings $s_1,\ \ldots s_n \in \Sigma^*$ can be simulated by the behaviour of ID on the corresponding set of inputs $\{\ s_1,\ \ldots s_n\ \}$. Once this is established, one can apply the known correctness of ID to establish the correctness of *IDS*. The IID algorithm of [Parekh et al. 1998] also presents a simulation

Algorithm 1 ID Learning Algorithm

Input: A live complete set $P \subseteq \Sigma^*$ and a teacher DFA A to answer membership queries $\alpha \in L(A)$?
Output: A DFA M equivalent to the target DFA A.

1. **begin**
2. //Perform Initialization
3. $i = 0$, $v_i = \lambda$, $V = \{\ v_i\ \}$,
4. $P' = P \cup \{\ d_0\ \}$, $T = P \cup \{f(\alpha, b) | (\alpha, b) \in P \times \Sigma\}$, $T' = T \cup \{\ d_0\ \}$
5. Construct function E_0 for $v_0 = \lambda$,
6. $E_0(d_0) = \emptyset$
7. $\forall \alpha \in T$
8. { pose the membership query "$\alpha \in L(A)$?"
9. **if** the teacher's response is yes
10. **then** $E_0(\alpha) = \{\lambda\}$
11. **else** $E_0(\alpha) = \emptyset$
12. **end if**
13. }
14. //Refine the partition of the set T'
15. **while** ($\exists \alpha, \beta \in P'$ and $b \in \Sigma$ such that $E_i(\alpha) = E_i(\beta)$ but $E_i(f(\alpha, b)) \neq E_i(f(\beta, b))$)
16. **do**
17. Let $\gamma \in E_i(f(\alpha, b)) \oplus E_i(f(\beta, b))$
18. $v_{i+1} = b\gamma$
19. $V = V \cup \{v_{i+1}\}$, $i = i + 1$
20. $\forall \alpha \in T_k$ pose the membership query "$\alpha v_i \in L(A)$?"
21. {
22. **if** the teacher's response is yes
23. **then** $E_i(\alpha) = E_{i-1}(\alpha) \cup \{v_i\}$
24. **else** $E_i(\alpha) = E_{i-1}(\alpha)$
25. **end if**
26. }
27. **end while**
28. //Construct the representation M of the target DFA A.
29. The states of M are the sets $E_i(\alpha)$, where $\alpha \in T$
30. The initial state q_0 is the set $E_i(\lambda)$
31. The accepting states are the sets $E_i(\alpha)$ where $\alpha \in T$ and $\lambda \in E_i(\alpha)$
32. The transitions of M are defined as follows:
33. $\forall \alpha \in P'$
34. **if** $E_i(\alpha) = \emptyset$
35. **then** add self loops on the state $E_i(\alpha)$ for all $b \in \Sigma$
36. **else** $\forall b \in \Sigma$ set the transition $\delta(E_i(\alpha), b) = E_i(f(\alpha, b))$
37. **end if**
38. **end.**

method for ID. However, it is easily shown that IID does not satisfy Proposition 3.2 or Simulation Theorem 3.4 below, and thus the two algorithms have different behaviour. The behavioural properties of ID that are needed to complete this correctness proof can be stated as follows.

2.1. Theorem.
(i) Let $P \subseteq \Sigma^*$ be a live complete set for a target DFA A containing λ. Then given P and A as input, the ID algorithm terminates and the automaton M returned by ID is the canonical automaton for $L(A)$.
(ii) Let $l \in \mathbb{N}$ be the maximum value of program variable i^{ID} given P and A. For all $0 \leq n \leq l$ and for all $\alpha \in T$,

$$E_n^{ID}(\alpha) = \{\ v_j^{ID} \mid 0 \leq j \leq n,\ \alpha v_j^{ID} \in L(A)\ \}.$$

Proof. (i) See [Angluin 1981] Theorem 3.
(ii) By induction on n.
Basis. Suppose $n = 0$. Then $v_0^{ID} = \lambda$. For any $\alpha \in T$, if $\alpha v_0^{ID} \in L(A)$ then $\alpha \in L(A)$ so $E_0^{ID}(\alpha) = \{\ v_0^{ID}\ \}$. If $\alpha v_0^{ID} \notin L(A)$ then $\alpha \notin L(A)$ so $E_0^{ID}(\alpha) = \emptyset$. Thus $E_0^{ID}(\alpha) = \{\ v_j^{ID} \mid 0 \leq j \leq 0,\ \alpha v_j^{ID} \in L(A)\ \}$.
Induction Step. Suppose $l \geq n > 0$. Consider any $\alpha,\ \beta \in P'$ and $b \in \Sigma$ such that $E_{n-1}^{ID}(\alpha) = E_{n-1}^{ID}(\beta)$ but $E_{n-1}^{ID}(f(\alpha, b)) \neq E_{n-1}^{ID}(f(\beta, b))$. Since $n - 1 < l$ then α, β and b exist. Then

$$E_{n-1}^{ID}(f(\alpha, b)) \oplus E_{n-1}^{ID}(f(\beta, b)) \neq \emptyset.$$

Consider any $\gamma \in E_{n-1}^{ID}(f(\alpha, b)) \oplus E_{n-1}^{ID}(f(\beta, b))$ and let $v_n^{ID} = b\gamma$. For any $\alpha \in T$, if $\alpha v_n^{ID} \in L(A)$ then $E_n^{ID}(\alpha) = E_{n-1}^{ID}(\alpha) \cup \{\ v_n^{ID}\ \}$ and if $\alpha v_n^{ID} \notin L(A)$ then $E_n^{ID}(\alpha) = E_{n-1}^{ID}(\alpha)$. So by the induction hypothesis $E_n^{ID}(\alpha) = \{\ v_j^{ID} \mid 0 \leq j \leq n,\ \alpha v_j^{ID} \in L(A)\ \}$.

3 Correctness of the *IDS* Algorithm

In this section we present our *IDS* incremental learning algorithm for DFA. In fact, we consider two versions of this algorithm, with and without prefix closure of the set of input strings. We then give a rigorous proof that both algorithms correctly learn an unknown DFA in the limit in the sense of [Gold 1967]

In Algorithm 2 we present the main *IDS* algorithm, and in Algorithms 3 and 4 we give its auxiliary algorithms for iterative partition refinement and automaton construction respectively.

The version of the *IDS* algorithm which appears in Algorithm 2 we term the *prefix free IDS* algorithm, due to lines 22 and 25. Notice that lines 23 and 26 of Algorithm 2 have been commented out. When these latter two lines are uncommented and instead lines 22 and 25 are commented out, we obtain a version of the *IDS* algorithm that we term *prefix closed IDS*. We will prove that both prefix closed and prefix free *IDS* learn correctly in the limit. However, in Section 4 we will show that they have quite different performance characteristics with respect to computation time and query types.

Algorithm 2 IDS Learning Algorithm

Input: A file $S = s_1, \ldots, s_l$ of input strings $s_i \in \Sigma^*$ and a teacher DFA A to answer membership queries $\alpha \in L(A)$?
Output: A sequence of DFA M_t for $t = 0, \ldots, l$ as well as the total number of membership queries and book keeping queries asked by the learner.

begin
  //Perform Initialization
  $i = 0$, $k = 0$, $t = 0$, $v_i = \lambda$, $V = \{\ v_i\ \}$
  //Process the empty string
  $P_0 = \{\lambda\}$, $P_0' = P_0 \cup \{d_0\}$, $T_0 = P_0 \cup \Sigma$
  $E_0(d_0) = \emptyset$
  $\forall \alpha \in T_0$ {
    pose the membership query "$\alpha \in L(A)$?", $bquery = bquery + 1$
    if the teacher's response is *yes*
    then $E_0(\alpha) = \{\lambda\}$
    else $E_0(\alpha) = \emptyset$
  }
  //Refine the partition of set T_0 as described in Algorithm 3
  //Construct the current representation M_0 of the target DFA
  //as described in Algorithm 4.

  //Process the file of examples.
  while $S \neq empty$ **do**
    read(S, α)
    mquery = mquery +1
    k = k+1, t = t+1
    $P_k = P_{k-1} \cup \{\alpha\}$
    // $P_k = P_{k-1} \cup Pref(\alpha)$ //prefix closure
    $P_k' = P_k \cup \{d_0\}$
    $T_k = T_{k-1} \cup \{\alpha\} \cup \{f(\alpha, b)\ \mid\ b \in \Sigma\}$
    // $T_k = P_k \cup \{f(\alpha, b)\ \mid\ \alpha \in P_k - P_{k-1}, b \in \Sigma\}$ //prefix closure
    $T_k' = T_k \cup \{d_0\}$
    $\forall \alpha \in T_k - T_{k-1}$
      {
        // Fill in the values of $E_i(\alpha)$ using membership queries:
        $E_i(\alpha) = \{v_j\ \mid\ 0 \leq j \leq i, \alpha v_j \in L(A)\}$
        $bquery = bquery + i$
      }
    // Refine the partition of the set T_k
    if α is consistent with M_{t-1}
    then $M_t = M_{t-1}$
    else construct M_t as described in Algorithm 4.
    }
end.

Algorithm 3 Refine Partition

1. **while** ($\exists \alpha, \beta \in P'_k$ and $b \in \Sigma$ such that $E_i(\alpha) = E_i(\beta)$ but $E_i(f(\alpha, b)) \neq E_i(f(\beta, b))$)
2. **do**
3. Let $\gamma \in E_i(f(\alpha, b)) \oplus E_i(f(\beta, b))$
4. $v_{i+1} = b\gamma$
5. $V = V \cup \{v_{i+1}\}$, $i = i + 1$
6. $\forall \alpha \in T_k$ pose the membership query "$\alpha v_i \in L(A)$?"
7. {
8. $bquery = bquery + 1$
9. **if** the teacher's response is *yes*
10. **then** $E_i(\alpha) = E_{i-1}(\alpha) \cup \{v_i\}$
11. **else** $E_i(\alpha) = E_{i-1}(\alpha)$
12. **end if**
13. }
14. **end while**

Algorithm 4 Automata Construction

1. The states of M_t are the sets $E_i(\alpha)$, where $\alpha \in T_k$
2. The initial state q_0 is the set $E_i(\lambda)$
3. The accepting states are the sets $E_i(\alpha)$ where $\alpha \in T_k$ and $\lambda \in E_i(\alpha)$
4. The transitions of M_t are defined as follows:
5. $\forall \alpha \in P'_k$
6. **if** $E_i(\alpha) = \emptyset$
7. **then** add self loops on the state $E_i(\alpha)$ for all $b \in \Sigma$
8. **else** $\forall b \in \Sigma$ set the transition $\delta(E_i(\alpha), b) = E_i(f(\alpha, b))$
9. **end if**
10. $\forall \beta \in T_k - P'_k$
11. **if** $\forall \alpha \in P'_k$ $E_i(\beta) \neq E_i(\alpha)$ **and** $E_i(\beta) \neq \emptyset$
12. **then** $\forall b \in \Sigma$ set the transition $\delta(E_i(\beta), b) = \emptyset$
13. **end if**

We will prove the correctness of the prefix free *IDS* algorithm first, since this proof is somewhat simpler, while the essential proof principles can also be applied to verify the prefix closed *IDS* algorithm. We begin an analysis of the correctness of prefix free *IDS* by confirming that the construction of hypothesis automata carried out by Algorithm 4 is well defined.

3.1. Proposition. *For each $t \geq 0$ the hypothesis automaton M_t constructed by the automaton construction Algorithm 4 after t input strings have been observed is a well defined DFA.*

Proof. We need to show that M_t is a well defined DFA $< \Sigma, Q, F, q_0, \delta >$. The input alphabet Σ for M_t is the same as the input alphabet for the target A. The state set Q is represented by the sets $E_i(\alpha)$, where $\alpha \in T_k$. The accepting state set F consists of all sets $E_i(\alpha)$, where $\lambda \in E_i(\alpha)$. By definition, $q_0 = E_i(\lambda)$ and since $P_k \neq \emptyset$ then $\lambda \in T_k$.

Finally for δ to be well defined function $\delta : \Sigma \times Q \to Q$ it must be uniquely defined for every state $E_i(\alpha)$, where $\alpha \in T_k$. So consider any $\alpha \in T_k$. By lines 8 and 12 of Algorithm 4, $\delta(E_i(\alpha), b)$ is defined for every $b \in \Sigma$. We need to show that $\delta(E_i(\alpha), b)$ is uniquely defined. So suppose $E_i(\alpha) = E_i(\beta)$, we must show that $\delta(E_i(\alpha), b) = \delta(E_i(\beta), b)$ for any $b \in \Sigma$.
(i) Suppose $\alpha \in P'_k$ and $\beta \in P'_k$ then by lines 1 to 12 of Algorithm 3, $E_i(f(\alpha,\ b)) = E_i(f(\beta,\ b))$ for all $b \in \Sigma$. Therefore by line 8 of Algorithm 4, $\delta(E_i(\alpha), b) = \delta(E_i(\beta), b)$.
(ii) Suppose $\alpha \in T_k - P'_k$ and $\beta \in P'_k$. If $E_i(\alpha) = E_i(\beta)$ then $\delta(E_i(\alpha), b)$ is already uniquely defined by (i) above.
(iii) Suppose $\alpha \in T_k - P'_k$ and $E_i(\alpha) \neq E_i(\beta)$ for any $\beta \in P'_k$ then by line 12 in Algorithm 4, $\delta(E_i(\alpha), b) = \emptyset$, so the transition is defined. To show that it is uniquely defined consider any $\beta \in T_k - P'_k$ such that $E_i(\alpha) = E_i(\beta)$. Then again by line 12 of Algorithm 4, $\delta(E_i(\beta), b) = \emptyset = \delta(E_i(\alpha), b)$.

Hence the hypothesis automaton M_t is a well defined DFA.

Proposition 3.1 establishes that Algorithm 2 will generate a sequence of well defined DFA. However, to show that this algorithm learns correctly, we must prove that this sequence of automata converges to the target automaton A given sufficient information about A. It will suffice to show that the behaviour of prefix free *IDS* can be simulated by the behaviour of ID, since ID is known to learn correctly given a live complete set of input strings (c.f. Theorem 2.1.(i)). The first step in this proof is to show that the sequences of sets of state names P_k^{IDS} and T_k^{IDS} generated by prefix free *IDS* converge to the sets P^{ID} and T^{ID} of ID.

3.2. Proposition. *Let $S = s_1, \ldots, s_l$ be any non-empty sequence of input strings $s_i \in \Sigma^*$ for prefix free IDS and let $P^{ID} = \{\ \lambda, s_1, \ldots, s_l\ \}$ be the corresponding input set for ID.*

(i) For all $0 \leq k \leq l$, $P_k^{IDS} = \{\ \lambda, s_1, \ldots, s_k\ \} \subseteq P^{ID}$.
(ii) For all $0 \leq k \leq l$, $T_k^{IDS} = P_k^{IDS} \cup \{\ f(\alpha, b) \mid \alpha \in P_k^{IDS}, b \in \Sigma\ \} \subseteq T^{ID}$.
(iii) $P_l^{IDS} = P^{ID}$ and $T_l^{IDS} = T^{ID}$.

Proof. Clearly (iii) follows from (i) and (ii). We prove (i) and (ii) by induction on k.
Basis. Suppose $k = 0$. (i) $P_0^{IDS} = \{\ \lambda\ \} \subseteq P^{ID}$. (ii) $T_0^{IDS} = \{\ \lambda\ \} \cup \Sigma \subseteq T^{ID}$.
Induction Step. Suppose $k > 0$. (i) By the induction hypothesis $P_{k-1}^{IDS} = \{\ \lambda,\ s_1,\ \ldots,\ s_{k-1}\ \} \subseteq P^{ID}$. So clearly

$$P_k^{IDS} = P_{k-1}^{IDS} \cup \{\ s_k\ \} = \{\ \lambda,\ s_1,\ \ldots,\ s_k\ \} \subseteq P^{ID}.$$

(ii) By Definition

$$T_k^{IDS} = T_{k-1}^{IDS} \cup \{\ s_k\ \} \cup \{\ f(\alpha,\ b)\ \mid\ \alpha \in P_k^{IDS} - P_{k-1}^{IDS},\ b \in \Sigma\ \}$$

$$= P_{k-1}^{IDS} \cup \{\ s_k\ \} \cup \{\ f(\alpha,\ b)\ \mid\ \alpha \in P_k^{IDS} - P_{k-1}^{IDS},\ b \in \Sigma\ \}$$

$$\cup \{\ f(\alpha,\ b)\ \mid\ \alpha \in P_{k-1}^{IDS},\ b \in \Sigma\ \}$$

by the induction hypothesis (ii)

$$= P_k^{IDS} \cup \{\ f(\alpha,\ b)\ \mid\ \alpha \in P_k^{IDS},\ b \in \Sigma\ \}.$$

Next we turn our attention to proving some fundamental loop invariants for Algorithm 2. Since this algorithm in turn calls the partition refinement Algorithm 3 then we have in effect a doubly nested loop structure to analyse. Clearly the two indexing counters k^{IDS} and i^{IDS} (in the outer and inner loops respectively) both increase on each iteration. However, the relationship between these two variables is not easily defined. Nevertheless, since both variables increase from an initial value of zero, we can assume the existence of a monotone re-indexing function that captures their relationship.

3.3. Definition. Let $S = s_1,\ \ldots,\ s_l$ be any non-empty sequence of strings $s_i \in \Sigma^*$. The *re-indexing function* $K^S : \mathbb{N} \to \mathbb{N}$ for prefix free *IDS* on input S is the unique monotonically increasing function such that for each $n \in \mathbb{N}$, $K^S(n)$ is the least integer m such that program variable k^{IDS} has value m while the program variable i^{IDS} has value n. Thus, for example, $K^S(0) = 0$. When S is clear from the context, we may write K for K^S.

With the help of such re-indexing functions we can express important invariant properties of the key program variables v_j^{IDS} and $E_n^{IDS}(\alpha)$, and via Proposition 3.2 their relationship to v_j^{ID} and $E_n^{ID}(\alpha)$. Corresponding to the doubly nested loop structure of Algorithm 2, the proof of Simulation Theorem 3.4 below makes use of a doubly nested induction argument.

3.4. Simulation Theorem. *Let $S = s_1,\ \ldots,\ s_l$ be any non-empty sequence of strings $s_i \in \Sigma^*$. For any execution of prefix free IDS on S there exists an execution of ID on $\{\ \lambda,\ s_1,\ \ldots,\ s_l\ \}$ such that for all $m \geq 0$:*
(i) For all $n \geq 0$ if $K(n) = m$ then:
(a) for all $0 \leq j \leq n$, $v_j^{IDS} = v_j^{ID}$,
(b) for all $0 \leq j < n$, $v_n^{IDS} \neq v_j^{IDS}$,

(c) for all $\alpha \in T_m^{IDS}$, $E_n^{IDS}(\alpha) = \{\ v_j^{IDS} \mid 0 \leq j \leq n,\ \alpha v_j^{IDS} \in L(A)\ \}$.
(ii) If $m > 0$ *then let* $p \in \mathbb{N}$ *be the greatest integer such that* $K(p) = m - 1$. *Then for all* $\alpha \in T_m^{IDS}$, $E_p^{IDS}(\alpha) = \{\ v_j^{IDS} \mid 0 \leq j \leq p,\ \alpha v_j^{IDS} \in L(A)\ \}$.
(iii) The m*th partition refinement of IDS terminates.*

Proof. By induction on m.
Basis. Suppose $m = 0$.
(i) We prove the result by subinduction on n.
Sub-basis. Suppose $n = 0$. Then $K(n) = m$.
(i.a) For ID and *IDS*, $v_0^{IDS} = \lambda = v_0^{ID}$.
(i.b) Holds vacuously since $n = 0$.
(i.c) Clearly $T_0^{IDS} = \{\ \lambda\ \} \cup \Sigma$. Consider any $\alpha \in T_0^{IDS}$. If $\alpha v_0^{IDS} \in L(A)$ then $\alpha \in L(A)$ and $E_0^{IDS}(\alpha) = \{\ v_0^{IDS}\ \}$. If $\alpha v_0^{IDS} \notin L(A)$ then $\alpha \notin L(A)$ and $E_0^{IDS}(\alpha) = \emptyset$. So

$$E_0^{IDS}(\alpha) = \{\ v_j^{IDS} \mid 0 \leq j \leq n,\ \alpha v_j^{IDS} \in L(A)\ \}.$$

Sub-Induction Step. Suppose $n > 0$ and $K(n) = m$.
(i.a) Consider any $\alpha,\ \beta \in {P'}_0^{IDS}$ and $b \in \Sigma$ such that $E_{n-1}^{IDS}(\alpha) = E_{n-1}^{IDS}(\beta)$ but $E_{n-1}^{IDS}(f(\alpha, b)) \neq E_{n-1}^{IDS}(f(\beta, b))$.

By Proposition 3.2.(i) $\alpha,\ \beta \in {P'}^{IDS}$, so by the sub-induction hypotheses (i.a) and (i.c) and Theorem 2.1.(ii) (since $K(n-1) = m = 0$), $E_{n-1}^{ID}(\alpha) = E_{n-1}^{ID}(\beta)$ but $E_{n-1}^{ID}(f(\alpha, b)) \neq E_{n-1}^{ID}(f(\beta, b))$. Also

$$E_{n-1}^{IDS}(f(\alpha, b)) \oplus E_{n-1}^{IDS}(f(\beta, b)) = E_{n-1}^{ID}(f(\alpha, b)) \oplus E_{n-1}^{ID}(f(\beta, b)) \neq \emptyset.$$

Let $\gamma \in E_{n-1}^{IDS}(f(\alpha, b)) \oplus E_{n-1}^{IDS}(f(\beta, b))$, then we can choose the same α, β, b and γ for an execution of ID so that $v_n^{IDS} = v_n^{ID} = b\gamma$. So by the sub-induction hypothesis (i.a) for all $0 \leq j \leq n$, $v_j^{IDS} = v_j^{ID}$.

(i.b) For α, β, b and γ as in (i.a) above either

$$\gamma \in E_{n-1}^{IDS}(f(\alpha, b))\ \ and\ \ \gamma \notin E_{n-1}^{IDS}(f(\beta, b)) \quad (1)$$

or

$$\gamma \notin E_{n-1}^{IDS}(f(\alpha, b))\ \ and\ \ \gamma \in E_{n-1}^{IDS}(f(\beta, b)) \quad (2).$$

Suppose (1) holds. Then $\gamma \in E_{n-1}^{IDS}(f(\alpha, b))$. So by the sub-induction hypothesis (i.c) for some $0 \leq x \leq n$, $\gamma = v_x^{IDS}$. Thus $v_n^{IDS} = b\gamma = bv_x^{IDS}$. Suppose for a contradiction that for some $0 \leq y < n$, $v_n^{IDS} = v_y^{IDS}$. Since $\alpha b v_x^{IDS} \in L(A)$ then

$$v_y^{IDS} = bv_x^{IDS} \in E_{n-1}^{IDS}(\alpha) \quad (3),$$

by sub-induction hypothesis (i.c). But by (1), $\gamma \notin E_{n-1}^{IDS}(f(\beta, b))$ so $v_x^{IDS} \notin E_{n-1}^{IDS}(f(\beta, b))$ hence $\beta b v_x^{IDS} \notin L(A)$ by sub-induction hypothesis (i.c). Thus $\beta v_y^{IDS} \notin L(A)$. So by sub-induction hypothesis (i.c),

$$v_y^{IDS} \notin E_{n-1}^{IDS}(\beta) \quad (4).$$

So by (3) and (4) $E_{n-1}^{IDS}(\alpha) \neq E_{n-1}^{IDS}(\beta)$. But this contradicts $E_{n-1}^{IDS}(\alpha) = E_{n-1}^{IDS}(\beta)$. By symmetry, if (2) holds a contradiction also arises. Thus for all $0 \leq j < n$, $v_j^{IDS} \neq v_n^{IDS}$.

(i.c) Consider any $\alpha \in T_0^{IDS}$. If $\alpha v_n^{IDS} \in L(A)$ then $E_n^{IDS}(\alpha) = E_{n-1}^{IDS}(\alpha) \cup \{ v_n^{IDS} \}$ and if $\alpha v_0^{IDS} \notin L(A)$ then $E_n^{IDS}(\alpha) = E_{n-1}^{IDS}(\alpha)$. So by the sub-induction hypothesis (i.c) since $K(n-1) = 0$ then

$$E_n^{IDS}(\alpha) = \{ \ v_j^{IDS} \ | \ 0 \leq j \leq n, \ \alpha v_j^{IDS} \in L(A) \ \}.$$

This completes the sub-induction proof of (i).

(ii) Holds trivially since $m \not> 0$.

(iii) Consider the 0-th refinement step in IDS. Clearly ${P'}_0^{IDS}$ is finite. For any $\alpha, \ \beta \in {P'}_0^{IDS}$ and $b \in \Sigma$ and $n \in \mathbb{N}$ such that $K(n) = 0$ and $E_{n-1}^{IDS}(\alpha) = E_{n-1}^{IDS}(\beta)$ but $E_{n-1}^{IDS}(f(\alpha, b)) \neq E_{n-1}^{IDS}(f(\beta, b))$, then

$$E_{n-1}^{IDS}(f(\alpha, b)) \oplus E_{n-1}^{IDS}(f(\beta, b)) \neq \emptyset.$$

So considering any $\gamma \in E_{n-1}^{IDS}(f(\alpha, b)) \oplus E_{n-1}^{IDS}(f(\beta, b))$ either

$$\gamma \in E_{n-1}^{IDS}(f(\alpha, b)) \ \ and \ \ \gamma \notin E_{n-1}^{IDS}(f(\beta, b)) \quad (5)$$

or

$$\gamma \notin E_{n-1}^{IDS}(f(\alpha, b)) \ \ and \ \ \gamma \in E_{n-1}^{IDS}(f(\beta, b)) \quad (6).$$

Let

$$v_n^{ID} = b\gamma \quad (7).$$

Suppose (5) holds. If $\alpha v_n^{IDS} \in L(A)$ then by (7) $\alpha b\gamma \in L(A)$ and $E_n^{IDS}(\alpha) = E_{n-1}^{IDS}(\alpha) \cup \{ \ v_n^{IDS} \ \}$. But by (5) $\gamma \notin E_{n-1}^{IDS}(f(\beta, b))$ so $\beta b\gamma \notin L(A)$. So $E_n^{IDS}(\beta) = E_{n-1}^{IDS}(\beta)$. By (i.b) and (i.c) $v_n^{IDS} \notin E_{n-1}^{IDS}(\alpha)$. So

$$E_n^{IDS}(\beta) = E_{n-1}^{IDS}(\beta) = E_{n-1}^{IDS}(\alpha) \neq E_n^{IDS}(\alpha).$$

By symmetry, the same result follows if (6) holds. Therefore on each iteration of the 0-th partition refinement loop, the number of such triples α, β and b strictly decreases. So the 0-th partition refinement loop must terminate.

Induction Step. Suppose $m > 0$.

(i) We prove the result by sub-induction on n.

Sub-basis. Suppose $n = 0$. Then $K(n) \neq m$ so the result holds trivially.

Sub-induction Step. Suppose $n > 0$ and $K(n) = m$.

(i.a) Suppose that n is the least integer such that $K(n) = m$, i.e. $K(n-1) = m-1$. Consider any $\alpha, \ \beta \in {P'}_m^{IDS}$ and $b \in \Sigma$ such that $E_{n-1}^{IDS}(\alpha) = E_{n-1}^{IDS}(\beta)$ but $E_{n-1}^{IDS}(f(\alpha, b)) \neq E_{n-1}^{IDS}(f(\beta, b))$. By Proposition 3.2.(i) $\alpha, \ \beta \in {P'}^{IDS}$ so by the

induction hypotheses (ii), (i.a) and Theorem 2.1.(ii), $E^{ID}_{n-1}(\alpha) = E^{ID}_{n-1}(\beta)$ but $E^{ID}_{n-1}(f(\alpha,b)) \neq E^{ID}_{n-1}(f(\beta,b))$. Furthermore

$$E^{IDS}_{n-1}(f(\alpha,b)) \oplus E^{IDS}_{n-1}(f(\beta,b)) = E^{ID}_{n-1}(f(\alpha,b)) \oplus E^{ID}_{n-1}(f(\beta,b)) \neq \emptyset.$$

Let $\gamma \in E^{IDS}_{n-1}(f(\alpha,b)) \oplus E^{IDS}_{n-1}(f(\beta,b))$ then we can choose the same α, β, b and γ for an execution of ID so that

$$v^{IDS}_n = b\gamma = v^{ID}_n.$$

By the sub-induction hypothesis (i.a) for all $0 \leq j \leq n$, $v^{IDS}_j = v^{ID}_j$.

Suppose that n is strictly greater than the least integer such that $K(n) = m$, i.e. $K(n-1) = m$. The proof is similar to above, but we use sub-induction hypothesis (i.a) instead of induction hypothesis (ii).

(i.b) The proof is similar to the proof of (i.b) in the subinduction step of the induction basis.

(i.c) For α, β, b and γ as in (i.a) above, suppose that n is the least integer such that $K(n) = m$, i.e. $K(n-1) = m-1$.

Consider any $\alpha \in T^{IDS}_m$. If $\alpha v^{IDS}_n \in L(A)$ then $E^{IDS}_n(\alpha) = E^{IDS}_{n-1}(\alpha) \cup \{\ v^{IDS}_n\ \}$, and if $\alpha v^{IDS}_n \notin L(A)$ then $E^{IDS}_n(\alpha) = E^{IDS}_{n-1}(\alpha)$. So by the induction hypothesis (ii)

$$E^{IDS}_n(\alpha) = \{\ v^{IDS}_j \mid 0 \leq j \leq n,\ \alpha v^{IDS}_j \in L(A)\ \}.$$

Suppose that n is greater than least integer such that $K(n) = m$, i.e. $K(n-1) = m$. The proof is similar to above but we use sub-induction hypothesis (i.c) instead of induction hypothesis (ii).

This completes the sub-induction proof of (i).

(ii) Let $p \in \mathbb{N}$ be the greatest integer such that $K(p) = m-1$. By the induction hypothesis (i.b) for all $\alpha \in T^{IDS}_{m-1}$

$$E^{IDS}_p(\alpha) = \{\ v^{IDS}_j \mid 0 \leq j \leq p,\ \alpha v^{IDS}_j \in L(A)\ \}.$$

and by line 3 of *IDS* Algorithm 2, for all $\alpha \in T^{IDS}_m - T^{IDS}_{m-1}$,

$$E^{IDS}_p(\alpha) = \{\ v^{IDS}_j \mid 0 \leq j \leq p,\ \alpha v^{IDS}_j \in L(A)\ \}.$$

So for all $\alpha \in T^{IDS}_m$

$$E^{IDS}_p(\alpha) = \{\ v^{IDS}_j \mid 0 \leq j \leq p,\ \alpha v^{IDS}_j \in L(A)\ \}.$$

(iii) Consider the m-th refinement step in prefix free *IDS*. The proof is similar to the proof of (iii) in the subinduction step of the induction basis.

Notice that in the statement of Theorem 3.4 above, since both ID and *IDS* are non-deterministic algorithms (due to the non-deterministic choice on line 17 of Algorithm 1 and line 3 of Algorithm 3), then we can only talk about the

existence of some correct simulation. Clearly there are also simulations of *IDS* by ID which are not correct, but this does not affect the basic correctness argument.

3.5. Corollary. *Let $S = s_1, \ldots, s_l$ be any non-empty sequence of strings $s_i \in \Sigma^*$. Any execution of prefix free IDS on S terminates with the program variable k^{IDS} having value l.*

Proof. Follows from Simulation Theorem 3.4.(iii) since clearly the while loop of Algorithm 2 terminates when the input sequence S is empty.

Using the detailed analysis of the invariant properties of the program variables P_k^{IDS} and T_k^{IDS} in Proposition 3.2 and v_j^{IDS} and $E_n^{IDS}(\alpha)$ in Simulation Theorem 3.4 it is now a simple matter to establish correctness of learning for the prefix free *IDS* Algorithm.

3.6. Correctness Theorem. *Let $S = s_1, \ldots, s_l$ be any non-empty sequence of strings $s_i \in \Sigma^*$ such that $\{ \lambda, s_1, \ldots, s_l \}$ is a live complete set for a target DFA A. Then prefix free IDS terminates on S and the hypothesis automaton M_l^{IDS} is a canonical representation of A.*

Proof. By Corollary 3.5, prefix free *IDS* terminates on S with the variable k^{IDS} having value l. By Simulation Theorem 3.4.(i) and Theorem 2.1.(ii), there exists an execution of ID on $\{ \lambda, s_1, \ldots, s_l \}$ such that $E_n^{IDS}(\alpha) = E_n^{ID}(\alpha)$ for all $\alpha \in T_l^{IDS}$ and any n such that $K(n) = l$. By Proposition 3.2.(iii), $T_l^{IDS} = T^{ID}$ and $P'^{IDS}_l = P'^{IDS}$. So letting M^{ID} be the canonical representation of A constructed by ID using $\{ \lambda, s_1, \ldots, s_l \}$ then M^{ID} and M_l^{IDS} have the same state sets, initial states, accepting states and transitions.

Our next result confirms that the hypothesis automaton M_t^{IDS} generated after t input strings have been read is consistent with all currently known observations about the target automaton. This is quite straightforward in the light of Simulation Theorem 3.4.

3.7. Compatibility Theorem. *Let $S = s_1, \ldots, s_l$ be any non-empty sequence of strings $s_i \in \Sigma^*$. For each $0 \leq t \leq l$ and each string $s \in \{ \lambda, s_1, \ldots, s_t \}$, the hypothesis automaton M_t^{IDS} accepts s if, and only if the target A does.*

Proof. By definition, M_t^{IDS} is compatible with A on $\{ \lambda, s_1, \ldots, s_t \}$ if, and only if, for each $0 \leq j \leq t$, $s_j \in L(A) \Leftrightarrow \lambda \in E_{i_t}^{IDS}(s_j)$, where i_t is the greatest integer such that $K(i_t) = t$ and the sets $E_{i_t}^{IDS}(\alpha)$ for $\alpha \in T_t^{IDS}$ are the states of M_t^{IDS}. Now $v_0^{IDS} = \lambda$. So by Simulation Theorem 3.4.(i).(c), if $s_j \in L(A)$ then $s_j v_0^{IDS} \in L(A)$ so $v_0^{IDS} \in E_{i_t}^{IDS}(s_j)$, i.e. $\lambda \in E_{i_t}^{IDS}(s_j)$, and if $s_j \notin L(A)$ then $s_j v_0^{IDS} \notin L(A)$ so $v_0^{IDS} \notin E_{i_t}^{IDS}(s_j)$, i.e. $\lambda \notin E_{i_t}^{IDS}(s_j)$.

Let us briefly consider the correctness of prefix closed *IDS*. We begin by observing that the non-sequential ID Algorithm 1 does not compute any prefix closure of input strings. Therefore, Proposition 3.2 does not hold for prefix closed *IDS*. In order to obtain a simulation between prefix closed *IDS* and ID we modify Proposition 3.2 to the following.

3.8. Proposition. *Let $S = s_1, \ldots, s_l$ be any non-empty sequence of input strings $s_i \in \Sigma^*$ for prefix closed IDS and let $P^{ID} = Pref(\{\ \lambda,\ s_1,\ \ldots,\ s_l\ \})$ be the corresponding input set for ID.*

(i) For all $0 \leq k \leq l$, $P_k^{IDS} = Pref(\{\ \lambda,\ s_1,\ \ldots,\ s_k\ \}) \subseteq P^{ID}$.
(ii) For all $0 \leq k \leq l$, $T_k^{IDS} = P_k^{IDS} \cup \{\ f(\alpha,\ b) \mid \alpha \in P_k^{IDS},\ b \in \Sigma\ \} \subseteq T^{ID}$.
(iii) $P_l^{IDS} = P^{ID}$ and $T_l^{IDS} = T^{ID}$.

Proof. Similar to the proof of Proposition 3.2.

We leave it to the reader, as an exercise, to make similar changes to Simulation Theorem 3.4 and Corollary 3.5, with the help of which one can establish the correctness of prefix closed *IDS*.

3.9. Correctness Theorem. *Let $S = s_1, \ldots, s_l$ be any non-empty sequence of strings $s_i \in \Sigma^*$ such that $\{\ \lambda,\ s_1,\ \ldots,\ s_l\ \}$ is a live complete set for a target DFA A. Then prefix closed IDS terminates on S and the hypothesis automaton M_l^{IDS} is a canonical representation of A.*

Proof. Exercise, following the proof of Theorem 3.6.

4 Empirical Performance Analysis

Little seems to have been published about the empirical performance and average time complexity of incremental learning algorithms for DFA in the literature. By the average time complexity of the algorithm we mean the average number of queries needed to completely learn a DFA of a given state space size. This question can be answered experimentally by randomly generating a large number of DFA with a given state space size, and randomly generating a sequence of query strings for each such DFA.

From the point of view of software engineering applications such as testing and model inference, we have found that it is important to distinguish between the two types of queries about the target automaton that are used by *IDS* during the learning procedure. On the one, hand the algorithm uses internally generated queries (we call these *book-keeping queries*) and on the other hand it uses queries that are supplied externally by the input file (we call these *membership queries*). From a software engineering applications viewpoint it seems important that the ratio of book-keeping to membership queries should be low. This allows membership queries to have the maximum influence in steering the learning process externally. The average query complexity of the *IDS* algorithm with respect to the numbers of book-keeping and membership queries needed for complete learning can also be measured by random generation of DFA and query strings. To measure each query type, Algorithm 2 has been instrumented with two integer variables *bquery* and *mquery* intended to track the total number of each type of query used during learning (lines 8, 20 and 32).

Since two variants of the *IDS* algorithm were identified, with and without prefix closure of input strings, it was interesting to compare the performance of each of these two variants according to the above two average complexity measures.

4.1 Experimental Procedure

To empirically measure the average time and query complexity of our two *IDS* algorithms, two experiments were set up. These measured:

(1) the average computation time needed to learn a randomly generated DFA (of a given state space size) using randomly generated membership queries, and

(2) the total number of membership and book-keeping queries needed to learn a randomly generated DFA (of a given state space size) using randomly generated membership queries.

We chose randomly generated DFA with state space sizes varying between 5 and 50 states, and an equiprobable distribution of transitions between states. No filtering was applied to remove dead states, so the average effective state space size was therefore somewhat smaller than the nominal state space size.

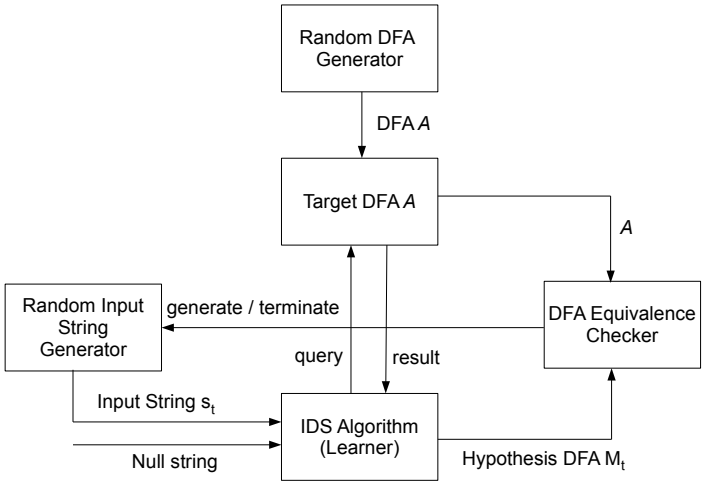

Fig. 1. Evaluation Framework

The experimental setup consisted of the following components:
(1) a random input string generator,
(2) a random DFA generator,
(3) an instance of the *IDS* Algorithm (prefix free or prefix closed) ,
(4) an automaton equivalence checker.

The architecture of our evaluation framework and the flow of data between these components are illustrated in Figure 1.

The purpose of the equivalence checker was to terminate the learning procedure as soon as the hypothesis automaton sequence had successfully converged

to the target automaton. There are several well known equivalence checking algorithms described in literature. These have runtime complexity ranging from quadratic to nearly linear execution times. We chose an algorithm with nearly linear time performance described in [Norton 2009]. This was to minimise the overhead of equivalence checking in the overall computation time. The *IDS* algorithms and the entire evaluation framework were implemented in Java. The performance of the input string and DFA generators is dependent on Java's Random class which generates pseudorandom numbers that depend upon a specific seed. To minimize the chance of generating the same pseudo random strings/automata again the seed was set to the system clock.

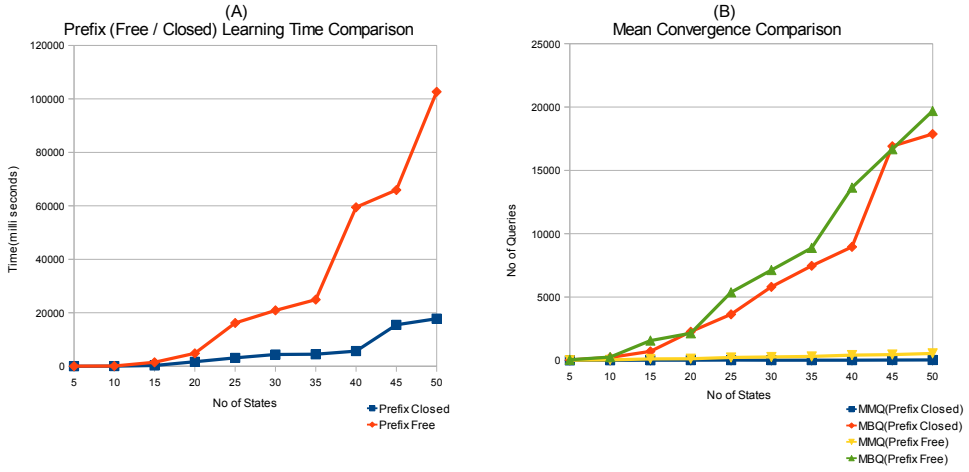

Fig. 2. Average Time Complexity

4.2 Results and Interpretation

The two graphs in Figure 2 illustrate the outcome of our experiments to measure the average time and average query complexity of both *IDS* algorithms, as described in Section 4.1.

Figure 2.A presents the results of estimating the average learning time for the prefix free and prefix closed *IDS* algorithms as a function of the state space size of the target DFA. For large state space sizes n, the data sets of randomly generated target DFA represent only a small fraction of all possible such DFA of size n. Therefore the two data curves are not smooth for large state space sizes. Nevertheless, there is sufficient data to identify some clear trends. The average learning time for prefix free *IDS* learning is substantially greater than corresponding time for prefix closed *IDS*, and this discrepancy increases with state space size. The reason would appear to be that prefix free *IDS* throws away data about the target DFA that must be regenerated randomly (since

input string queries are generated at random). The average time complexity for prefix free *IDS* learning seems to grow approximately quadratically, while the average time complexity for prefix closed *IDS* learning appears to grow almost linearly within the given data range. From this viewpoint, prefix-closed *IDS* appears to be the superior algorithm.

Figure 2.B presents the results of estimating the average number of membership queries and book-keeping queries as a function of the state space size of the target DFA. Again, we have compared prefix-closed with prefix free *IDS* learning. Allowance must also be made for the small data set sizes for large state space values. We can see that membership queries grow approximately linearly with the increase in state space size, while book-keeping queries grow approximately quadratically, at least within the data ranges that we considered. There appears to be a small but significant decrease in the number of both book-keeping and membership queries used by the prefix-closed *IDS* algorithm. The reason for this appears to be similar to the issues identified for average time complexity. Prefix closure seems to be an efficient way to gather data about the target DFA. From the viewpoint of software engineering applications discussed in Section 1, now prefix free *IDS* appears to be preferable. This is because the decreasing ratio of book-keeping to membership queries improves the possibility to direct the learning process using externally generated queries (e.g. from a model checker).

5 Conclusions

We have presented two versions of the *IDS* algorithm which is an incremental algorithm for learning DFA in polynomial time. We have given a rigorous proof that both algorithms correctly learn in the limit. Finally we have presented the results of an empirical study of the average time and query complexity of *IDS*. These empirical results suggest that *IDS* is well suited to applications in software engineering, where an incremental approach that allows externally generated online queries is needed. This conclusion is further supported in [Meinke and Sindhu 2011], where we have evaluated the *IDS* algorithm for learning based testing of reactive systems, and shown that it leads to error discovery up to 4000 times faster than using non-incremental learning.

We gratefully acknowledge financial support for this research from the Higher Education Commission (HEC) of Pakistan, the Swedish Research Council (VR) and the European Union under project HATS FP7-231620.

References

[Angluin 1981] D. Angluin, A note on the number of queries needed to identify regular languages, Information and Control, 51:76-87, 1981.

[Angluin 1987] D. Angluin, Learning regular sets from queries and counterexamples, Information and Computation, 75:87-106, 1987.

[Bohlin and Jonsson 2008] T. Bohlin and B. Jonsson, Regular Inference for Communication Protocol Entities, Tech. Report 2008-024, Dept. of Information Technology, Uppsala University, 2008.

[Clarke et. al. 2002] E. Clarke, A. Gupta, J. Kukula, O. Strichman, SAT-based Abstraction Refinement Using ILP and Machine Learning, in: Proc. 21st International Conference On Computer Aided Verification (CAV'02) 2002.

[Dupont 1996] Incremental regular inference, pp 222-237 in: L. Miclet and C. Huguera (eds) Proceedings of the Third ICGI-96, LNAI 1147, Springer, 1996.

[Gold 1967] E.M. Gold, Language identification in the limit, Information and Control 10(5):447-474, 1967.

[Lang 1992] K.J. Lang, Random DFA's can be approximately learned from sparse uniform examples, in: proc. 5th ACM workshop on Computational Learning Theory, pp45-52, 1992.

[Luecker 2006] M. Luecker, Learning meets verification, pp 127-151 in: F.S. de Boer et al. (eds), FMCO, LNCS 4709, Springer, 2006.

[Meinke 2004] K. Meinke, Automated Black-Box Testing of Functional Correctness using Function Approximation, pp 143-153 in: G. Rothermel (ed) *Proc. ACM SIGSOFT Int. Symp. on Software Testing and Analysis*, ISSTA 2004, Software Engineering Notes 29 (4), ACM Press, 2004.

[Meinke, Niu 2010] K. Meinke, F. Niu, A Learning-based Approach to Unit Testing of Numerical Software, in Proc. ICTSS 2010, Lecture Notes in Computer Science, Springer Verlag, 2010.

[Meinke and Sindhu 2011] K. Meinke and M. Sindhu, *Incremental Learning-based Testing for Reactive Systems*, pp 134-151 in M. Gogolla and B.Wolff (eds) Proc Fifth Int. Conf. on Tests and Proofs (TAP2011) LNCS 6706, Springer Verlag, 2011.

[Norton 2009] Algorithms for Testing Equivalence of Finite State Automata, with a Grading Tool for JFLAP, MS Project Report, Rochester Institute of Technology-Department of Computer Science. http://www.cs.rit.edu/~dan0337, 2009.

[Oncina and Garcia 1992] J. Oncina and P. Garcia, Inferring regular languages in polynomial update time, pp 49-61 in: N. Perez de la Blanca, A. Sanfeliu and E. Vidal (eds), Pattern Recognition and Image Analysis, Vol. 1 of Series in Machine Perception and Artificial Intelligence, World Scientific, 1992.

[Parekh et al. 1998] R.G. Parekh, C. Nichitiu and V.G. Honavar. A polynomial time incremental algorithm for regular grammar inference, in: Proc. Fourth ICGI-98, Springer, 1998.

[Peled et al. 1999] D. Peled, M.Y. Vardi, M. Yannakakis, Black-box Checking, in J. Wu et al. (eds), *Formal Methods for Protocol Engineering and Distributed Systems*, FORTE/PSTV, 225-240, Beijing, 1999, Kluwer.

[Porat, Feldman 1991] S. Porat and J. Feldman, Learning automata from ordered examples, Machine Learning 7:109-138, 1991.

[Raffelt et al. 2008] H. Raffelt , B. Steffen and T. Margaria, Dynamic Testing Via Automata Learning, pp 136-152 in: Hardware and Software: Verification and Testing, Lecture Notes in Computer Science, Vol. 4899, Springer, 2008.

Appendix B

Paper 2 (Incremental Learning-based Testing for Reactive Systems)

Incremental Learning-Based Testing for Reactive Systems

K. Meinke, M. A. Sindhu

School of Computer Science and Communication,
Royal Institute of Technology, 100-44 Stockholm, Sweden,
karlm@nada.kth.se, sindhu@csc.kth.se

Abstract. We show how the paradigm of learning-based testing (LBT) can be applied to automate specification-based black-box testing of reactive systems. Since reactive systems can be modeled as Kripke structures, we introduce an efficient incremental learning algorithm IKL for such structures. We show how an implementation of this algorithm combined with an efficient model checker such as NuSMV yields an effective learning-based testing architecture for automated test case generation (ATCG), execution and evaluation, starting from temporal logic requirements.

1 Introduction

A heuristic approach to automated test case generation (ATCG) from formal requirements specifications known as *learning-based testing* (LBT) was introduced in Meinke [9] and Meinke and Niu [11]. Learning-based testing is an iterative approach to automate specification-based black-box testing. It encompasses both test case generation, execution and evaluation (the oracle step). The aim of this approach is to automatically generate a large number of high-quality test cases by combining a model checking algorithm with an optimised model inference algorithm. For procedural programs, [11] has shown that LBT can significantly outperform random testing in the speed with which it finds errors in a system under test (SUT).

In this paper we consider how the LBT approach can be applied to a quite different class of SUTs, namely *reactive systems*. Conventionally, reactive systems are modeled as *Kripke structures* and their requirements are usually specified using a *temporal logic* (see e.g. [6]). To learn and test such models efficiently, we therefore introduce a new learning algorithm IKL (Incremental Kripke Learning) for Kripke structures. We show that combining the IKL algorithm for model inference together with an efficient temporal logic model checker such as NuSMV yields an effective LBT architecture for reactive systems. We evaluate the effectiveness of this testing architecture by means of case studies.

In the remainder of Section 1 we discuss the general paradigm of LBT, and specific requirements on learning. In Section 2 we review some essential mathematical preliminaries. In Section 3, we consider the technique of bit-sliced learning of Kripke structures. In Section 4, we present a new incremental learning algorithm IKL for Kripke structures that uses distinguishing sequences, bit-slicing,

and lazy partition refinement. In Section 5 we present a complete LBT architecture for reactive systems testing. We evaluate this architecture by means of case studies in Section 6. Finally, in Section 7 we draw some conclusions.

1.1 Learning-Based Testing

Several previous works, (for example Peled et al. [16], Groce et al. [8] and Raffelt et al. [17]) have considered a combination of learning and model checking to achieve testing and/or formal verification of reactive systems. Within the model checking community the verification approach known as *counterexample guided abstraction refinement* (CEGAR) also combines learning and model checking, (see e.g. Clarke et al. [5]). The LBT approach can be distinguished from these other approaches by: (i) an emphasis on testing rather than verification, and (ii) use of *incremental learning algorithms* specifically chosen to make testing more effective and scalable (c.f. Section 1.2).

The basic LBT paradigm requires three components:

(1) a (black-box) *system under test* (SUT) S,

(2) a *formal requirements specification Req* for S, and

(3) a *learned model* M of S.

Now (1) and (2) are common to all specification-based testing, and it is really (3) that is distinctive. Learning-based approaches are *heuristic iterative methods* to automatically generate a sequence of test cases. The heuristic approach is based on *learning a black-box system using tests as queries*.

An LBT algorithm iterates the following four steps:

(Step 1) Suppose that n test case inputs $i_1, \ldots, i_n$ have been executed on S yielding the system outputs $o_1, \ldots, o_n$. The n input/output pairs $(i_1, o_1), \ldots, (i_n, o_n)$ are synthesized into a learned model M_n of S using an *incremental learning algorithm* (see Section 1.2). This step involves *generalization* from the given data, (which represents an incomplete description of S) to all possible data. It gives the possibility to predict previously unseen errors in S during Step 2.

(Step 2) The system requirements *Req* are satisfiability checked against the learned model M_n derived in Step 1 (aka. *model checking*). This process searches for a *counterexample* i_{n+1} to the requirements.

(Step 3) The counterexample i_{n+1} is executed as the next test case on S, and if S terminates then the output o_{n+1} is obtained. If S fails this test case (i.e. the pair (i_{n+1}, o_{n+1}) does not satisfy *Req*) then i_{n+1} was a *true negative* and we proceed to Step 4. Otherwise S passes the test case i_{n+1} so the model M_n was inaccurate, and i_{n+1} was a *false negative*. In this latter case, the effort of executing S on i_{n+1} is not wasted. We return to Step 1 and apply the learning algorithm once again to $n+1$ pairs $(i_1, o_1), \ldots, (i_{n+1}, o_{n+1})$ to infer a refined model M_{n+1} of S.

(Step 4) We terminate with a true negative test case (i_{n+1}, o_{n+1}) for S.

Thus an LBT algorithm iterates Steps 1 . . . 3 until an SUT error is found (Step 4) or execution is terminated. Possible criteria for termination include a bound on the maximum testing time, or a bound on the maximum number of test cases to be executed.

This iterative approach to TCG yields a sequence of increasingly accurate models M_0, M_1, M_2, ..., of S. (We can take M_0 to be a minimal or even empty model.) So, with increasing values of n, it becomes more and more likely that satisfiability checking in Step 2 will produce a true negative if one exists. If Step 2 does not produce any counterexamples at all then to proceed with the iteration, we must construct the next test case i_{n+1} by some other method, e.g. randomly.

1.2 Efficient Learning Algorithms

As has already been suggested in Step 1 of Section 1.1, for LBT to be effective at finding errors, it is important to use the right kind of learning algorithm. A good learning algorithm should maximise the opportunity of the satisfiability algorithm in Step 2 to find a true counterexample i_{n+1} to the requirements *Req* as soon as possible.

An automata learning algorithm L is said to be *incremental* if it can produce a sequence of hypothesis automata A_0, A_1, ... which are approximations to an unknown automata A, based on a sequence of information (queries and results) about A. The sequence A_0, A_1, ... must finitely converge to A, at least up to behavioural equivalence. In addition, the computation of each new approximation A_{i+1} by L should reuse as much information as possible about the previous approximation A_i (e.g. equivalences between states). Incremental learning algorithms are necessary for efficient learning-based testing of reactive systems for two reasons.

(1) Real reactive systems may be too big to be completely learned and tested within a feasible timescale. This is due to the typical complexity properties of learning and satisfiability algorithms.

(2) Testing of specific requirements such as use cases may not require learning and analysis of the entire reactive system, but only of a fragment that implements the requirement *Req*.

For testing efficiency, we also need to consider the *type of queries* used during learning. The overhead of SUT execution to answer a membership query during learning can be large compared with the execution time of the learning algorithm itself (see e.g. [3]). So membership queries should be seen as "expensive". Therefore, as many queries (i.e. test cases) as possible should be derived from model checking the hypothesis automaton, since these are all based on checking the requirements *Req*. Conversely as few queries as possible should be derived for reasons of internal book-keeping by the learning algorithm (e.g. for achieving congruence closure prior to automaton construction). Book-keeping queries make no reference to the requirements *Req*, and therefore can only uncover an SUT error by accident. Ideally, *every* query would represent a relevant and interesting requirements-based test case. In fact, if the percentage of internally

generated book-keeping queries is too high then model checking becomes almost redundant. In this case we might think that LBT becomes equivalent to random testing. However [18] shows that this is not the case. Even without model checking, LBT achieves better functional coverage than random testing.

In practise, most of the well-known classical regular inference algorithms such as L* (Angluin [2]) or ID (Angluin [1]) are designed for complete rather incremental learning. Among the much smaller number of known incremental learning algorithms, we can mention the RPNII algorithm (Dupont [7]) and the IID algorithm (Parekh et al. [15]) which learn Moore automata, and the CGE algorithm (Meinke [10]) which learns Mealy automata. To our knowledge, no incremental algorithm for learning Kripke structures has yet been published in the literature. Thus the IKL algorithm, and its application to testing represent novel contributions of our paper.

2 Mathematical Preliminaries and Notation

Let Σ be any set of symbols then Σ^* denotes the set of all finite strings over Σ including the empty string ε. The length of a string $\alpha \in \Sigma^*$ is denoted by $|\alpha|$ and $|\varepsilon| = 0$. For strings $\alpha, \beta \in \Sigma^*$, $\alpha \,.\, \beta$ denotes their concatenation.

For $\alpha, \beta, \gamma \in \Sigma^*$, if $\alpha = \beta\gamma$ then β is termed a *prefix* of α and γ is termed a *suffix* of α. We let $Pref(\alpha)$ denote the prefix closure of α, i.e. the set of all prefixes of α. We can also apply prefix closure pointwise to any set of strings. The *set difference operation* between two sets U, V, denoted by $U - V$, is the set of all elements of U which are not members of V. The *symmetric difference* operation on pairs of sets is defined by $U \oplus V = (U - V) \cup (V - U)$.

A *deterministic finite automaton* (DFA) is a five-tuple $A =< \Sigma, Q, F, q_0, \delta >$ where: Σ is the input alphabet, Q is the state set, $F \subseteq Q$ is the accepting state set and $q_0 \in Q$ is the starting state. The state transition function of A is a mapping $\delta : Q \times \Sigma \to Q$ with the usual meaning, and can be inductively extended to a mapping $\delta^* : Q \times \Sigma^* \to Q$ where $\delta^*(q,\ \varepsilon) = q$ and $\delta^*(q,\ \sigma_1,\ \ldots \sigma_{n+1}) = \delta(\delta^*(q,\ \sigma_1,\ \ldots \sigma_n),\ \sigma_{n+1})$. Since input strings can be used to name states, given a distinguished *dead state* d_0 (from which no accepting state can be reached) we define *string concatenation modulo the dead state* d_0, $f : \Sigma^* \cup \{d_0\} \times \Sigma \to \Sigma^* \cup \{d_0\}$, by $f(d_0,\ \sigma) = d_0$ and $f(\alpha,\ \sigma) = \alpha \,.\, \sigma$ for $\alpha \in \Sigma^*$. This function is used for automaton learning in Section 4. The *language* $L(A)$ *accepted by* A is the set of all strings $\alpha \in \Sigma^*$ such that $\delta^*(q_0,\ \alpha) \in F$. A language $L \subseteq \Sigma^*$ is accepted by a DFA if and only if, L is *regular*, i.e. L can be defined by a regular grammar.

A generalisation of DFA to multi-bit outputs on states is given by deterministic Kripke structures.

2.1. Definition. Let $\Sigma = \{\ \sigma_1,\ \ldots,\ \sigma_n\ \}$ be a finite input alphabet. By a k-bit *deterministic Kripke structure* A we mean a five-tuple

$$A = (\ Q_A,\ \Sigma,\ \delta_A : Q_A \times \Sigma \to Q_A,\ q_A^0,\ \lambda_A : Q_A \to \mathbb{B}^k\ )$$

where Q_A is a state set, δ_A is the state transition function, q_A^0 is the initial state and λ_A is the output function. As before we let $\delta_A^* : Q_A \times \Sigma^* \to Q_A$ denote the iterated state transition function, where $\delta_A^*(q,\ \varepsilon) = q$ and $\delta_A^*(q,\ \sigma_1,\ \ldots,\ \sigma_{i+1}) = \delta_A(\delta_A^*(q,\ \sigma_1,\ \ldots,\ \sigma_i),\ \sigma_{i+1})$. Also we let $\lambda_A^* : \Sigma^* \to \mathbb{B}^k$ denote the iterated output function $\lambda_A^*(\sigma_1,\ \ldots,\ \sigma_i) = \lambda_A(\delta_A^*(q_A^0,\ \sigma_1,\ \ldots,\ \sigma_i))$.

If A is a Kripke structure then the *minimal subalgebra* $Min(A)$ of A is the unique subalgebra of A which has no proper subalgebra. (We implicitly assume that all input symbols $\sigma \in \Sigma$ are constants of A so that $Min(A)$ has a non-trivial state set.) Note that a 1-bit deterministic Kripke structure A is isomorphic to the DFA $A' = (\ Q_A,\ \Sigma,\ \delta_A : Q_A \times \Sigma \to Q_A,\ q_A^0,\ F_{A'}\ )$, where $F_{A'} \subseteq Q_A$ and $\lambda_A(q) = \mathit{true}$ if, and only if $q \in F_{A'}$.

3 Bit-Sliced Learning of Kripke Structures

We will establish a precise basis for learning k-bit Kripke structures using regular inference algorithms for DFA. The approach we take is to bit-slice the output of a k-bit Kripke structure A into k individual 1-bit Kripke structures $A_1,\ \ldots,\ A_k$, which are learned in parallel as DFA by some regular inference algorithm. The k inferred DFA $B_1,\ \ldots,\ B_k$ are then recombined using a subdirect product construction to obtain a Kripke structure that is behaviourally equivalent to A.

This approach has three advantages: (1) We can make use of *any* regular inference algorithm to learn the individual 1-bit Kripke structures A_i. Thus we have access to the wide range of known regular inference algorithms. (2) We can reduce the total number of book-keeping queries by *lazy book-keeping*. This technique maximises re-use of book-keeping queries among the 1-bit structures A_i. In Section 4, we illustrate this technique in more detail. (3) We can learn just those bits which are necessary to test a specific temporal logic requirement. This *abstraction technique* improves the scalability of testing.

It usually suffices to learn automata up to behavioural equivalence.

3.1. Definition. Let A and B be k-bit Kripke structures over a finite input alphabet Σ. We say that A and B are *behaviourally equivalent*, and write $A \equiv B$ if, and only if, for every finite input sequence $\sigma_1,\ \ldots,\ \sigma_i \in \Sigma^*$ we have

$$\lambda_A^*(\ \sigma_1,\ \ldots,\ \sigma_i\ ) = \lambda_B^*(\ \sigma_1,\ \ldots,\ \sigma_i\ ).$$

Clearly, by the isomorphism identified in Section 2 between 1-bit Kripke structures and DFA, for such structures we have $A \equiv B$ if, and only if, $L(A') = L(B')$. Furthermore, if $Min(A)$ is the minimal subalgebra of A then $Min(A) \equiv A$.

Let us make precise the concept of bit-slicing a Kripke structure.

3.2. Definition. Let A be a k-bit Kripke structure over a finite input alphabet Σ,

$$A = (\ Q_A,\ \Sigma,\ \delta_A : Q_A \times \Sigma \to Q_A,\ q_A^0,\ \lambda_A : Q_A \to \mathbb{B}^k\ ).$$

For each $1 \leq i \leq k$ define the *i-th projection* A_i of A to be the 1-bit Kripke structure where

$$A_i = (\ Q_A,\ \Sigma,\ \delta_A : Q_A \times \Sigma \to Q_A,\ q_A^0,\ \lambda_{A_i} : Q_A \to \mathbb{B}\ ),$$

and $\lambda_{A_i}(q) = \lambda_A(q)_i$, i.e. $\lambda_{A_i}(q)$ is the i-th bit of $\lambda_A(q)$.

A family of k individual 1-bit Kripke structures can be combined into a single k-bit Kripke structure using the following subdirect product construction. (See e.g. [13] for a general definition of subdirect products and their universal properties.)

3.3. Definition. Let $A_1,\ \ldots,\ A_k$ be a family of 1-bit Kripke structures,

$$A_i = (\ Q_i,\ \Sigma,\ \delta_i : Q_i \times \Sigma \to Q_i,\ q_i^0,\ \lambda_i : Q \to \mathbb{B}\ )$$

for $i = 1,\ \ldots,\ k$. Define the *product Kripke structure*

$$\prod_{i=1}^{k} A_i = (\ Q,\ \Sigma,\ \delta : Q \times \Sigma \to Q,\ q^0,\ \lambda : Q \to \mathbb{B}^k\ ),$$

where $Q = \prod_{i=1}^k Q_i = Q_1 \times \ldots \times Q_k$ and $q^0 = (\ q_1^0,\ \ldots,\ q_k^0\ )$. Also

$$\delta(q_1,\ \ldots,\ q_k,\ \sigma) = (\ \delta_1(q_1,\ \sigma),\ \ldots,\ \delta_k(q_k,\ \sigma)\ ),$$

$$\lambda(q_1,\ \ldots,\ q_k) = (\ \lambda_1(q_1),\ \ldots,\ \lambda_k(q_k)\ ).$$

Associated with the direct product $\prod_{i=1}^k A_i$ we have i-th *projection mapping*

$$proj_i : Q_1 \times \ldots \times Q_k \to Q_i, \qquad proj_i(q_1,\ \ldots,\ q_k) = q_i,\ 1 \leq i \leq k$$

Let $Min(\ \prod_{i=1}^k A_i\ )$ be the minimal subalgebra of $\prod_{i=1}^k A_i$.

The reason for taking the minimal subalgebra of the direct product $\prod_{i=1}^k A_i$ is to avoid the state space explosion due to a large number of unreachable states in the direct product itself. The state space size of $\prod_{i=1}^k A_i$ grows exponentially with k. On the other hand since most of these states are unreachable from the initial state, then from the point of view of behavioural analysis these states are irrelevant. Note that this minimal subalgebra can be computed in linear time from its components A_i (w.r.t. state space size).

As is well known from universal algebra, the i-th projection mapping $proj_i$ is a homomorphism.

3.4. Proposition. *Let $A_1,\ \ldots,\ A_k$ be any minimal 1-bit Kripke structures.*
(i) *For each $1 \leq i \leq k$, $proj_i : Min(\ \prod_{i=1}^k A_i\ ) \to A_i$ is an epimorphism, and hence $Min(\ \prod_{i=1}^k A_i\ )$ is a subdirect product of the A_i.*

(ii) $Min(\ \prod_{i=1}^k A_i\ ) \equiv \prod_{i=1}^k A_i$.

Proof. (i) Immediate since the A_i are minimal. (ii) Follows from $Min(A) \equiv A$.

The following theorem justifies bit-sliced learning of k-bit Kripke structures using conventional regular inference methods for DFA.

3.5. Theorem. *Let A be a k-bit Kripke structure over a finite input alphabet Σ. Let $A_1, \ldots, A_k$ be the k individual 1-bit projections of A. For any 1-bit Kripke structures $B_1, \ldots, B_k$, if, $A_1 \equiv B_1 \,\&\, \ldots \,\&\, A_k \equiv B_k$ then*

$$A \equiv Min(\prod_{i=1}^{k} B_i\ ).$$

Proof. Use Proposition 3.4.

4 Incremental Learning for Kripke Structures

In this section we present a new algorithm for incremental learning of Kripke structures. We will briefly discuss its correctness and termination properties, although a full discussion of these is outside the scope of this paper and is presented elsewhere in [12]. Our algorithm applies bit-slicing as presented in Section 3, and uses distinguishing sequences and lazy partition refinement for regular inference of the 1-bit Kripke structures. The architecture of the IKL algorithm consists of a main learning algorithm and two sub-procedures for lazy partition refinement and automata synthesis. Distinguishing sequences were introduced in Angluin [1] as a method for learning DFA.

Algorithm 1 is the main algorithm for bit-sliced incremental learning. It learns a sequence $M_1, \ldots, M_l$ of n-bit Kripke structures that successively approximate a single n-bit Kripke structure A, which is given as the teacher. In LBT, the teacher is always the SUT.

The basic idea of Algorithm 1 is to construct in parallel a family $E^1_{i_1}, \ldots, E^n_{i_n}$ of n different equivalence relations on the same set T_k of state names. For each equivalence relation $E^j_{i_j}$, a set V_j of distinguishing strings is generated iteratively to split pairs of equivalence classes in $E^j_{i_j}$ until a congruence is achieved. Then a quotient DFA M^j can be constructed from the partition of T_k by $E^j_{i_j}$. The congruences are constructed so that $E^j_i \subseteq E^j_{i+1}$ and thus the IKL algorithm is incremental, and fully reuses information about previous approximations, which is efficient.

Each n-bit Kripke structure M_t is constructed using synthesis algorithm 3, as a subdirect product of n individual quotient DFA $M^1, \ldots, M^n$ (viewed as 1-bit Kripke structures). When the IKL algorithm is applied to the problem of LBT, the input strings $s_i \in \Sigma^*$ to IKL are generated as counterexamples to correctness (i.e. test cases) by executing a model checker on the approximation M_{t-1} with respect to some requirements specification ϕ expressed in temporal logic. If no counterexamples to ϕ can be found in M_{t-1} then s_i is randomly chosen, taking care to avoid all previously used input strings.

Algorithm 2 implements lazy partition refinement, to extend $E^1_{i_1}, \ldots, E^n_{i_n}$ from being equivalence relations on states to being a family of congruences with respect to the state transition functions $\delta_1, \ldots, \delta_n$ of $M^1, \ldots, M^n$.

Algorithm 1 IKL: Incremental Kripke Structure Learning Algorithm

Input: A file $S = s_1, \ldots, s_l$ of input strings $s_i \in \Sigma^*$ and a Kripke structure A with n-bit output as teacher to answer queries $\lambda_A^*(s_i) = ?$
Output: A sequence of Kripke structures M_t with n-bit output for $t = 0, \ldots, l$.

1. **begin**
2. //Perform Initialization
3. **for** c = 1 to n **do** { $i_c = 0$, $v_{i_c} = \varepsilon$, $V_c = \{v_{i_c}\}$ }
4. $k = 0$, $t = 0$,
5. $P_0 = \{\varepsilon\}$, $P_0' = P_0 \cup \{d_0\}$, $T_0 = P_0 \cup \Sigma$
6. //Build equivalence classes for the dead state d_0
7. **for** c = 1 to n **do** { $E_0^c(d_0) = \emptyset$ }
8. //Build equivalence classes for input strings of length zero and one
9. $\forall \alpha \in T_0$ {
10. $(b_1, \ldots, b_n) = \lambda_A^*(\alpha)$
11. **for** c = 1 to n **do**
12. **if** b_c **then** $E_{i_c}^c(\alpha) = \{v_{i_c}\}$ **else** $E_{i_c}^c(\alpha) = \emptyset$
13. }
14. //Refine the initial equivalence relations $E_0^1, \ldots, E_0^n$
15. //into congruences using Algorithm 2
16.
17. //Synthesize an initial Kripke structure M_0 approximating A
18. //using Algorithm 3.
19.
20. //Process the file of examples.
21. **while** $S \neq empty$ **do** {
22. read(S, α)
23. k = k+1, t = t+1
24. $P_k = P_{k-1} \cup Pref(\alpha)$ //prefix closure
25. $P_k' = P_k \cup \{d_0\}$
26. $T_k = T_{k-1} \cup Pref(\alpha) \cup \{\alpha \,.\, b \mid \alpha \in P_k - P_{k-1}, b \in \Sigma\}$ //for prefix closure
27. $T_k' = T_k \cup \{d_0\}$
28. $\forall \alpha \in T_k - T_{k-1}$ {
29. **for** c = 1 to n **do** $E_0^c(\alpha) = \emptyset$ //initialise new equivalence class $E_0^c(\alpha)$
30. **for** j = 0 to i_c **do** {
31. // Consider adding distinguishing string $v_j \in V_c$
32. // to each new equivalence class $E_j^c(\alpha)$
33. $(b_1, \ldots, b_n) = \lambda_A^*(\alpha \,.\, v_j)$
34. **if** b_c **then** $E_j^c(\alpha) = E_j^c(\alpha) \cup \{ v_j \}$
35. }
36. }
37. //Refine the current equivalence relations $E_{i_1}^1, \ldots, E_{i_n}^n$
38. // into congruences using Algorithm 2
39.
40. **if** α is consistent with M_{t-1}
41. **then** $M_t = M_{t-1}$
42. **else** synthesize Kripke structure M_t using Algorithm 3.
43. }
44. **end.**

Algorithm 2 Lazy Partition Refinement

1. **while** ($\exists\ 1 \leq c \leq n, \exists \alpha, \beta \in P'_k$ and $\exists \sigma \in \Sigma$ such that $E^c_{i_c}(\alpha) = E^c_{i_c}(\beta)$ but $E^c_{i_c}(f(\alpha,\ \sigma)) \neq E^c_{i_c}(f(\beta,\ \sigma))$ **do** {
2. //Equivalence relation $E^c_{i_c}$ is not a congruence w.r.t. δ_c
3. //so add a new distinguishing sequence.
4. **Choose** $\gamma \in E^c_{i_c}(f(\alpha,\ \sigma)) \oplus E^c_{i_c}(f(\beta,\ \sigma))$
5. $v = \sigma\ .\ \gamma$
6. $\forall \alpha \in T_k$ {
7. $(b_1,\ \ldots,\ b_n) = \lambda^*_A(\alpha\ .\ v)$
8. **for** c = 1 to n **do** {
9. **if** $E^c_{i_c}(\alpha) = E^c_{i_c}(\beta)$ and $E^c_{i_c}(f(\alpha,\ \sigma)) \neq E^c_{i_c}(f(\beta,\ \sigma))$ **then** {
10. // Lazy refinement of equivalence relation $E^c_{i_c}$
11. $i_c = i_c + 1$, $v_{i_c} = v$, $V_c = V_c \cup \{v_{i_c}\}$
12. **if** b_c **then** $E^c_{i_c}(\alpha) = E^c_{i_c-1}(\alpha) \cup \{v_{i_c}\}$ **else** $E^c_{i_c}(\alpha) = E^c_{i_c-1}(\alpha)$
13. }
14. }
15. }

Algorithm 3 Kripke Structure Synthesis

1. **for** c = 1 to n **do** {
2. // Synthesize the quotient DFA (1-bit Kripke structure) M^c
3. The states of M^c are the sets $E^c_{i_c}(\alpha)$, where $\alpha \in T_k$
4. Let $q^c_0 = E^c_{i_c}(\varepsilon)$
5. The accepting states are the sets $E^c_{i_c}(\alpha)$ where $\alpha \in T_k$ and $\varepsilon \in E^c_{i_c}(\alpha)$
6. The transition function δ_c of M^c is defined as follows:
7. $\forall \alpha \in P'_k$ {
8. **if** $E^c_{i_c}(\alpha) = \emptyset$ **then** $\forall b \in \Sigma$ { let $\delta_c(E^c_{i_c}(\alpha), b) = E^c_{i_c}(\alpha)$ }
9. **else** $\forall b \in \Sigma$ { $\delta_c(E^c_{i_c}(\alpha), b) = E^c_{i_c}(\alpha\ .\ b)$ }
10. }
11. $\forall \beta \in T_k - P'_k$ {
12. **if** $\forall \alpha \in P'_k$ { $E^c_{i_c}(\beta) \neq E^c_{i_c}(\alpha)$ } and $E^c_{i_c}(\beta) \neq \emptyset$ **then**
13. $\forall b \in \Sigma$ { $\delta_c(E^c_{i_c}(\beta), b) = \emptyset$ }
14. }
15. // Compute M_t in linear time as a subdirect product of the M^c
16. $M_t = Min(\ \prod_{c=1}^{n} M^c\ )$

Thus line 1 searches for congruence failure in any one of the equivalence relations $E^1_{i_1}, \ldots, E^n_{i_n}$. In lines 6-14 we apply lazy partition refinement. This technique implies reusing the new distinguishing string v wherever possible to refine each equivalence relation $E^j_{i_j}$ that is not yet a congruence. On the other hand, any equivalence relation $E^j_{i_j}$ that is already a congruence is not refined, even though the result b_j of the new query $\alpha \,.\, v$ might add some new information to M^j. This helps minimise the total number of partition refinement queries (cf. Section 1.2).

Algorithm 3 implements model synthesis. First, each of the n quotient DFA $M^1, \ldots, M^n$ are constructed. These, reinterpreted as 1-bit Kripke structures, are then combined in linear time as a subdirect product to yield a new n-bit approximation M_t to A (c.f. Section 3).

4.1 Correctness and Termination of the IKL algorithm.

The sequence $M_1, \ldots, M_l$ of hypothesis Kripke structures which are incrementally generated by IKL can be proven to finitely converge to A up to behavioural equivalence, for sufficiently large l. The key to this observation lies in the fact that we can identify a finite set of input strings such that the behavior of A is completely determined by its behaviour on this finite set.

Recall that for a DFA $A =< \Sigma, Q, F, q_0, \delta >$ a state $q \in Q$ is said to be *live* if for some string $\alpha \in \Sigma^*$, $\delta^*(q,\ \alpha) \in F$. A finite set $C \subseteq \Sigma^*$ of input strings is said to be *live complete* for A if for every reachable live state $q \in Q$ there exists a string $\alpha \in C$ such that $\delta^*(q_0,\ \alpha) = q$. More generally, given a finite collection $A_1, \ldots, A_k$ of DFA, then $C \subseteq \Sigma^*$ is *live complete* for $A_1, \ldots, A_k$ if, and only if, for each $1 \leq i \leq k$, C is a live complete set for A_i. Clearly, for every finite collection of DFA there exists at least one live complete set of strings.

4.1.1. Theorem. *Let A be a k-bit Kripke structure over a finite input alphabet Σ. Let $A_1, \ldots, A_k$ be the k individual 1-bit projections of A. Let $C = \{\ s_1, \ldots, s_l\ \} \subseteq \Sigma^*$ be a live complete set for $A_1, \ldots, A_k$. The IKL algorithm terminates on C and for the final hypothesis structure M_l we have*

$$M_l \equiv A.$$

Proof. See [12].

5 A Learning-Based Testing Architecture using IKL.

Figure 1 depicts an architecture for learning-based testing of reactive systems by combining the IKL algorithm of Section 4 with a model checker for Kripke stuctures and an oracle. In this case we have chosen to use the NuSMV model checker (see e.g. Cimatti et al. [4]), which supports the satisfiability analysis of Kripke structures with respect to both linear temporal logic (LTL) and computation tree logic (CTL) [6].

To understand this architecture, it is useful to recall the abstract description of learning-based testing as an iterative process, given in Section 1.1. Following the account of Section 1.1, we can assume that at any stage in the testing process we have an inferred Kripke structure M_n produced by the IKL algorithm from previous testing and learning. Test cases will have been produced as counterexamples to correctness by the model checker, and learning queries will have been produced by the IKL algorithm during partition refinement. (Partition refinement queries are an example of what we termed book-keeping queries in Section 1.2.)

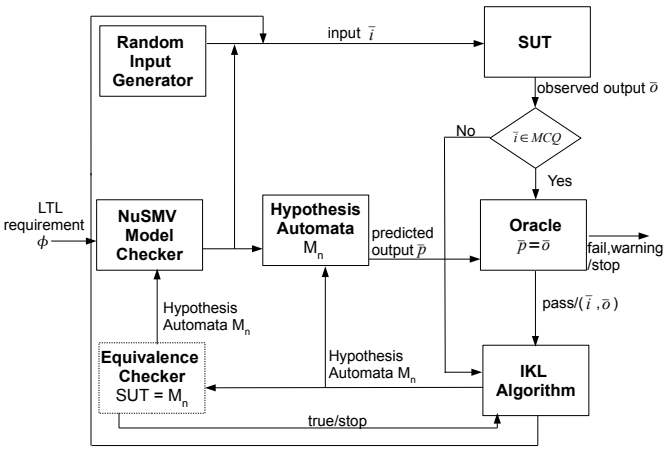

Fig. 1. A Learning-Based Testing Architecture using the IKL algorithm.

In Figure 1, the output M_n of the IKL algorithm is passed to an *equivalence checker*. Since this architectural component is not normally part of an LBT framework, we should explain its presence carefully. We are particularly interested in benchmarking the performance of LBT systems, both to compare their performance with other testing methodologies, and to make improvements to existing LBT systems. (See Section 6.) In realistic testing situations, we do not anticipate that an entire SUT can be learned in a feasible time (c.f. the discussion in Section 1.2). However, for benchmarking with the help of smaller case studies (for which complete learning is feasible) it is useful to be able to infer the earliest time at which we can say that testing is complete. Obviously testing must be complete at time t_{total} when we have learned the entire SUT (c.f. Section 6). Therefore the equivalence checker allows us to compute the time t_{total} simply to conduct benchmarking studies. (Afterwards the equivalence checker is removed.) The equivalence checker compares the current Kripke structure M_n with the SUT. A positive result from this equivalence test stops all further learning and testing after one final model check. The algorithm we use has

been adapted from the quasi-linear time algorithm for DFA equivalence checking described in [14] and has been extended to deal with k-bit Kripke structures.

In Figure 1, the inferred model M_n is passed to a model checker, together with a user requirement represented as a temporal logic formula ϕ. This formula is constant during a particular testing experiment. The model checker attempts to identify at least one counterexample to ϕ in M_n as an *input sequence* $\overline{i}$. If ϕ is a safety formula then this input sequence will usually be finite $\overline{i} = i_1, \ldots, i_k$. If ϕ is a liveness formula then this input sequence $\overline{i}$ may be finite or infinite. Recall that infinite counterexamples to liveness formulas can be represented as infinite sequences of the form $\overline{x}\,\overline{y}^{\omega}$. In the case that $\overline{i} = \overline{x}\,\overline{y}^{\omega}$ then $\overline{i}$ is truncated to a finite initial segment that would normally include the handle $\overline{x}$ and at least one execution of the infinite loop $\overline{y}^{\omega}$, such as $\overline{i} = \overline{x}\,\overline{y}$ or $\overline{i} = \overline{x}\,\overline{y}\,\overline{y}$. Observing the failure of an infinite test case is of course impossible. The LBT architecture implements a compromise solution that runs the truncated sequence only, in finite time, and issues a warning rather than a fail verdict.

Note that if the next input sequence $\overline{i}$ cannot be constructed either by partition refinement or by model checking then in order to proceed with iterative testing and learning, another way to generate $\overline{i}$ must be found. (See the discussion in Section 1.1.) One simple solution, shown in Figure 1, is to use a *random input sequence generator* for $\overline{i}$, taking care to discard any previously used sequences.

Thus from one of three possible sources (partition refinement, model checking or randomly) a new input sequence $\overline{i} = i_1, \ldots, i_k$ is constructed. Figure 1 shows that if $\overline{i}$ is obtained by model checking then the current model M_n is applied to $\overline{i}$ to compute a *predicted output* $\overline{p} = p_1, \ldots, p_k$ for the SUT that can be used for the oracle step. However, this is not possible if $\overline{i}$ is random or a partition refinement since then we do not know whether $\overline{i}$ is a counterexample to ϕ. Nevertheless, in all three cases, the input sequence $\overline{i}$ is passed to the SUT and executed to yield an *actual* or *observed output sequence* $\overline{o} = o_1, \ldots, o_k$.

The final stage of this iterative testing architecture is the *oracle step*. Figure 1 shows that if a predicted output $\overline{p}$ exists (i.e. the input sequence $\overline{i}$ came from model checking) then actual output $\overline{o}$ and the predicted output $\overline{p}$ are both passed to an *oracle* component. This component implements the Boolean test $\overline{o} = \overline{p}$. If this equality test returns *true* and the test case $\overline{i} = i_1, \ldots, i_k$ was originally a finite test case then we can conclude that the test case $\overline{i}$ is definitely *failed*, since the behaviour $\overline{p}$ is by construction a counterexample to the correctness of ϕ. If the equality test returns *true* and the test case $\overline{i}$ is finitely truncated from an infinite test case (a counterexample to a liveness requirement) then the verdict is weakened to a *warning*. This is because the most we can conclude is that we have not yet seen any difference between the observed behaviour $\overline{o}$ and the incorrect behaviour $\overline{p}$. The system tester is thus encouraged to consider a potential SUT error.

On the other hand if $\overline{o} \neq \overline{p}$, or if no output prediction $\overline{p}$ exists then it is quite difficult to issue an immediate verdict. It may or may not be the case that the observed output $\overline{o}$ is a counterexample to the correctness of ϕ. In some cases the syntactic structure of ϕ is simple enough to semantically evaluate the formula

ϕ on the fly with its input and output variables bound to $\bar{i}$ and $\bar{o}$ respectively. However, sometimes this is not possible since the semantic evaluation of ϕ also refers to global properties of the automaton. Ultimately, this is not a problem for our approach, since M_{n+1} is automatically updated with the output behaviour $\bar{o}$. Model checking M_{n+1} later on will confirm $\bar{o}$ as an error if this is the case.

5.1 Correctness and Termination of the LBT Architecture.

It is important to establish that the LBT architecture always terminates, at least in principle. Furthermore, the SUT coverage obtained by this testing procedure is complete, in the sense that if the SUT contains any counterexamples to correctness then a counterexample *will* be found by the testing architecture. When the SUT is too large to be completely learned in a feasible amount of time, this completeness property of the testing architecture still guarantees that there is no *bias* in testing so that one could somehow never discover an SUT error. Failure to find an error in this case is purely a consequence of insufficient testing time.

The termination and correctness properties of the LBT architecture depend on the following correctness properties of its components:

(i) the IKL algorithm terminates and correctly learns the SUT given a finite set C of input strings which is live complete (c.f. Theorem 4.1.1);

(ii) the model checking algorithm used by NuSMV is a terminating decision procedure for the validity of LTL formulas;

(iii) each input string $i \in \Sigma^*$ is generated with non-zero probability by the random input string generator.

5.1.1. Theorem. *Let A be a k-bit Kripke structure over an input alphabet Σ.*

(i) The LBT architecture (with equivalence checker) terminates with probability 1.0*, and for the final hypothesis structure M_l we have*

$$M_l \equiv A.$$

(ii) If there exists a (finite or infinite) input string $\bar{i}$ over Σ which witnesses that an LTL requirement ϕ is not valid for A, then model checking will eventually find such a string $\bar{i}$ and the LBT architecture will generate a test fail or test warning message after executing $\bar{i}$ as a test case on A.

Proof. (i) Clearly by Theorem 4.1.1, the IKL algorithm will learn the SUT A up to behavioural equivalence, given as input a live complete set C for the individual 1-bit projections $A_1, \ldots, A_k$ of A. Now, we cannot be sure that the strings generated by model checking counterexamples and partition refinement queries alone constitute a live complete set C for $A_1, \ldots, A_k$. However, these sets of queries are complemented by random queries. Since a live complete set is finite, and every input string is randomly generated with non-zero probability, then with probability 1.0 the IKL algorithm will eventually obtain a live complete set and converge. At this point, equivalence checking the final hypothesis structure M_l with the SUT will succeed and the LBT architecture will terminate.

(ii) Suppose there is at least one (finite or infinite) counterexample string $\overline{i}$ over Σ to the validity of an LTL requirement ϕ for A. In the worst case, by part (i), the LBT architecture will learn the entire structure of A. Since the model checker implements a terminating decision procedure for validity of LTL formulas, it will return a counterexample $\overline{i}$ from the final hypothesis structure M_l, since by part (i), $M_l \equiv A$ and A has a counterexample. For such $\overline{i}$, comparing the corresponding predicted output $\overline{p}$ from M_l and the observed output $\overline{o}$ from A we must have $\overline{p} = \overline{o}$ since $M_l \equiv A$. Hence the testing architecture will issue a fail or warning message.

6 Case Studies and Performance Benchmarking

In order to evaluate the effectiveness of the LBT architecture described in Section 5, we conducted a number of testing experiments on two SUT case studies, namely an 8 state cruise controller and a 38 state 3-floor elevator model[1].

For each SUT case study we chose a collection of safety and liveness requirements that could be expressed in linear temporal logic (LTL). For each requirement we then injected an error into the SUT that violated this requirement and ran a testing experiment to discover the injected error. The injected errors all consisted of *transition mutations* obtained by redirecting a transition to a wrong state. This type of error seems quite common in our experience.

There are a variety of ways to measure the performance of a testing system such as this. One simple measure that we chose to consider was to record the *first time* t_{first} at which an error was discovered in an SUT, and to compare this with the *total time* t_{total} required to completely learn the SUT. (So $t_{first} \leq t_{total}$.) This measure is relevant if we wish to estimate the benefit of using incremental learning instead of complete learning.

Because some random queries are almost always present in each testing experiment, the performance of the LBT architecture has a degree of variation. Therefore, for the same correctness formula and injected error, we ran each experiment ten times to try to average out these variations in performance. This choice appeared adequate to obtain a representative average. Subsections 6.1 and 6.2 below set out the results obtained for each case study.

6.1 The Cruise Controller Model

The cruise controller model we chose as an SUT is an 8 state 5-bit Kripke structure with an input alphabet of 5 symbols. Figure 2 shows its structure[2]. The four requirements shown in Table 1 consist of: (1,2) two requirements on speed maintenance against obstacles in cruise mode, and (3,4) disengaging cruise mode by means of the brake and gas pedals. To gain insight into the LBT

[1] Our testing platform was based on a PC with a 1.83 GHz Intel Core 2 duo processor and 4GB of RAM running Windows Vista.

[2] The following binary data type encoding is used. Modes: 00 = manual, 01 = cruise, 10 = disengaged. Speeds: 00 = 0, 01 = 1, 10 = 2.

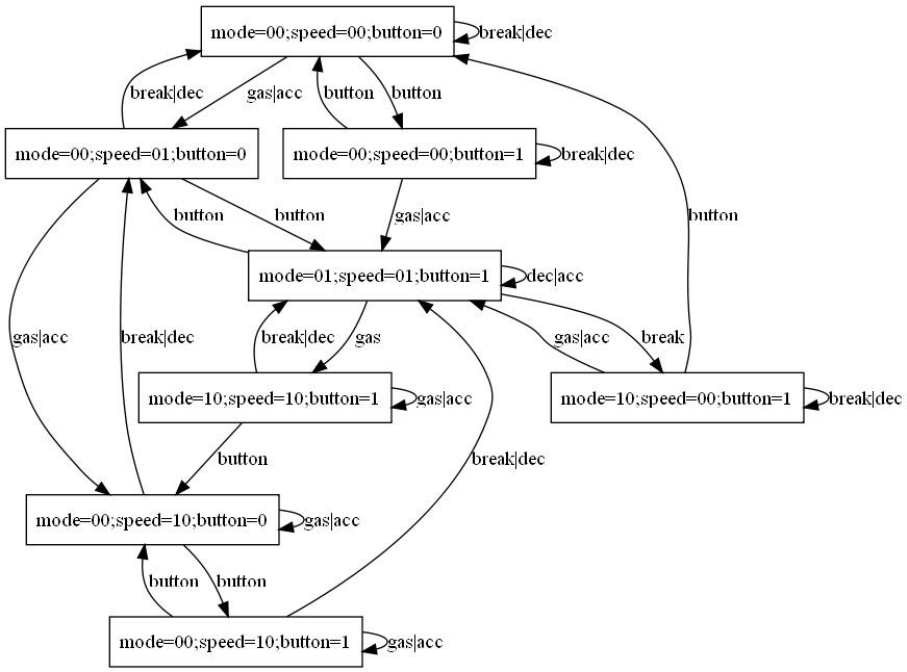

Fig. 2. The cruise controller SUT.

Req 1	G(mode = cruise & speed = 1 & in = dec -> X(speed = 1))
Req 2	G(mode = cruise & speed = 1 & in = acc -> X(speed = 1))
Req 3	G(mode = cruise & in = brake -> X(mode = disengaged))
Req 4	G(mode = cruise & in = gas -> X(mode = disengaged))

Table 1. Cruise Controller Requirements as LTL formulas.

Requirement	t_{first} (sec)	t_{total} (sec)	MCQ_{first}	MCQ_{total}	PQ_{first}	PQ_{total}	RQ_{first}	RQ_{total}
Req 1	3.5	21.5	3.2	24.3	7383	30204	8.2	29.3
Req 2	2.3	5.7	5.5	18.2	8430	27384	10.4	23.1
Req 3	2.3	16.0	1.7	33.7	6127	34207	6.8	38.8
Req 4	2.9	6.1	4.7	20.9	7530	24566	10.4	20.9

Table 2. LBT performance for Cruise Controller Requirements.

architecture performance, Table 2 shows average figures at times t_{first} and t_{total} for the numbers:

(i) MCQ_{first} and MCQ_{total} of model checker generated test cases,
(ii) PQ_{first} and PQ_{total} of partition refinement queries,
(iii) RQ_{first} and RQ_{total} of random queries.

In Table 2, columns 2 and 3 show that the times required to first discover an error in the SUT are between 14% and 47% of the total time needed to completely learn the SUT. The large query numbers in columns 6 and 7 show that partition refinement queries dominate the total number of queries. Columns 8 and 9 show that the number of random queries used is very low, of and of the same order of magnitude as the number of model checking queries (columns 4 and 5). Thus partition refinement queries and model checker generated test cases come quite close to achieving a live complete set, although they do not completely suffice for this (c.f. Section 4.1).

6.2 The Elevator Model

The elevator model we chose as an SUT is a 38 state 8-bit Kripke structure with an input alphabet of 4 symbols. Figure 3 shows its condensed structure as a hierarchical statechart. The six requirements shown in Table 3 consist of

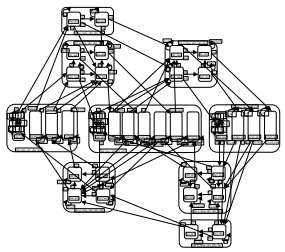

Fig. 3. The 3-floor elevator SUT (condensed Statechart).

Req 1	G(Stop -> (@1 | @2 | @3))
Req 2	G(!Stop -> cl)
Req 3	G(Stop & X(!Stop) -> X(!cl))
Req 4	G(Stop & @1 & cl & in=c1 & X(@1) -> X(!cl))
Req 5	G(Stop & @2 & cl & in=c2 & X(@2) -> X(!cl))
Req 6	G(Stop & @3 & cl & in=c3 & X(@3) -> X(!cl))

Table 3. Elevator Requirements as LTL formulas.

Requirement	t_{first} (sec)	t_{total} (sec)	MCQ_{first}	MCQ_{total}	PQ_{first}	PQ_{total}	RQ_{first}	RQ_{total}
Req 1	0.34	1301.3	1.9	81.7	1574	729570	1.9	89.5
Req 2	0.49	1146	3.9	99.6	2350	238311	2.9	98.6
Req 3	0.94	525	1.6	21.7	6475	172861	5.7	70.4
Req 4	0.052	1458	1.0	90.3	15	450233	0.0	91
Req 5	77.48	2275	1.2	78.3	79769	368721	20.5	100.3
Req 6	90.6	1301	2.0	60.9	129384	422462	26.1	85.4

Table 4. LBT performance for Elevator Requirements.

requirements that: (1) the elevator does not stop between floors, (2) doors are closed when in motion, (3) doors open upon reaching a floor, and (4, 5, 6) closed doors can be opened by pressing the same floor button when stationary at a floor.

Table 4 shows the results of testing the requirements of Table 3. These results confirm several trends seen in Table 2. However, they also show a significant increase in the efficiency of using incremental learning, since the times required to first discover an error in the SUT are now between 0.003% and 7% of the total time needed to completely learn the SUT. These results are consistent with observations of [12] that the convergence time of IKL grows quadratically with state space size. Therefore incremental learning gives a more scalable testing method than complete learning.

7 Conclusions

We have presented a novel incremental learning algorithm for Kripke structures, and shown how this can be applied to learning-based testing of reactive systems. Using two case studies of reactive systems, we have confirmed our initial hypothesis of Section 1.2, that incremental learning is a more scalable and efficient method of testing than complete learning. These results are consistent with similar results for LBT applied to procedural systems in [11].

Further research could be carried out to improve the performance of the architecture presented here. For example the performance of the oracle described in Section 5 could be improved to yield a verdict even for random and partition queries, at least for certain kinds of LTL formulas. Further research into scalable learning algorithms would be valuable for dealing with large hypothesis automata. The question of learning-based coverage has been initially explored in [18] but further research here is also needed.

We gratefully acknowledge financial support for this research from the Swedish Research Council (VR), the Higher Education Commission (HEC) of Pakistan, and the European Union under project HATS FP7-231620.

References

1. D. Angluin. A note on the number of queries needed to identify regular languages. *Information and Control*, 51(1):76–87, October 1981.
2. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(1):87–106, November 1987.
3. T. Bohlin and B. Jonsson. Regular inference for communication protocol entities. Technical Report 2008-024, Dept. of Information Technology, Uppsala University, 2008.
4. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model verifier. In *Proceeding of International Conference on Computer-Aided Verification (CAV'99)*, number 1633 in LNCS. Springer, 1999.
5. E. Clarke, A. Gupta, J. Kukula, and O. Strichman. Sat-based abstraction refinement using ilp and machine learning. In *Proc. 21st International Conference On Computer Aided Verification (CAV'02)*, 2002.
6. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.
7. P. Dupont. Incremental regular inference. In *Proceedings of the Third ICGI-96*, number 1147 in LNAI, 1996.
8. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. *Logic Journal of the IGPL*, 14(5):729–744, 2006.
9. K. Meinke. Automated black-box testing of functional correctness using function approximation. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 143–153, New York, NY, USA, 2004. ACM.
10. K. Meinke. Cge: A sequential learning algorithm for mealy automata. In *Proc. Tenth Int. Colloq. on Grammatical Inference (ICGI 2010)*, number 6339 in LNAI, pages 148–162. Springer, 2010.
11. K. Meinke and F. Niu. A learning-based approach to unit testing of numerical software. In *Proc. Twenty Second IFIP Int. Conf. on Testing Software and Systems (ICTSS 2010)*, number 6435 in LNCS, pages 221–235. Springer, 2010.
12. K. Meinke and M. Sindhu. Correctness and performance of an incremental learning algorithm for kripke structures. Technical report, School of Computer Science and Communication, Royal Institute of Technology, Stockholm, 2010.
13. K. Meinke and J.V. Tucker. Universal algebra. In *Handbook of Logic in Computer Science: Volume 1*, pages 189–411. Oxford University Press, 1993.
14. D.A. Norton. Algorithms for testing equivalence of finite state automata, with a grading tool for jflap. Technical report, Rochester Institute of Technology, Department of Computer Science, 2009.
15. R.G. Parekh, C. Nichitiu, and V.G. Honavar. A polynomial time incremental algorithm for regular grammar inference. In *Proc. Fourth Int. Colloq. on Grammatical Inference (ICGI 98)*, LNAI. Springer, 1998.
16. D. Peled, M.Y. Vardi, and M. Yannakakis. Black-box checking. In *Formal Methods for Protocol Engineering and Distributed Systems FORTE/PSTV*, pages 225–240. Kluwer, 1999.
17. H. Raffelt, B. Steffen, and T. Margaria. Dynamic testing via automata learning. In *Hardware and Software: Verification and Testing*, number 4899 in LNCS, pages 136–152. Springer, 2008.
18. N. Walkinshaw, K. Bogdanov, J. Derrick, and J. Paris. Increasing functional coverage by inductive testing: a case study. In *Proc. Twenty Second IFIP Int. Conf. on Testing Software and Systems (ICTSS 2010)*, number 6435 in LNCS, pages 126–141. Springer, 2010.