Algorithms and Tools for Learning-based Testing of Reactive Systems

MUDDASSAR AZAM SINDHU

Doctoral Thesis
Stockholm, Sweden 2013
Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie doktorandexamen i datalogi tisdagen den 16 april 2013 klockan 10.00 i F3, Lindstedtsvägen 26, Kungl Tekniska högskolan, Stockholm.

© Muddassar Azam Sindhu, April 2013

Tryck: E-Print AB
Dedication

To

Lala Jee

(My grand father Ch. Faqeer Muhammad Sindhu who passed away on Aug 06, 2005 at the age of 105 years and always wished and prayed for his grandson to reach the zenith.)
Abstract

In this thesis we investigate the feasibility of learning-based testing (LBT) as a viable testing methodology for reactive systems. In LBT, a large number of test cases are automatically generated from black-box requirements for the system under test (SUT) by combining an incremental learning algorithm with a model checking algorithm. The integration of the SUT with these algorithms in a feedback loop optimizes test generation using the results from previous outcomes. The verdict for each test case is also created automatically in LBT.

To realize LBT practically, existing algorithms in the literature both for complete and incremental learning of finite automata were studied. However, limitations in these algorithms led us to design, verify and implement new incremental learning algorithms for DFA and Kripke structures. On the basis of these algorithms we implemented an LBT architecture in a practical tool called LBTest which was evaluated on pedagogical and industrial case studies.

The results obtained from both types of case studies show that LBT is an effective methodology which discovers errors in reactive SUTs quickly and can be scaled to test industrial applications. We believe that this technology is easily transferrable to industrial users because of its high degree of automation.
Acknowledgements

All praise be to Allah the Almighty and His countless blessings be upon His beloved Prophet Muhammad (Peace Be Upon Him). I bow down before my Allah with all my humility to thank Him for enabling me to complete this PhD thesis.

The journey of this PhD to me was like a trek through the Himalayas with several lows and highs along the way to reach its highest peak. The vagaries of such a trail look diminished if one is accompanied by a *guru* accustomed to trudging the trail. In my case, the guru turned out to be Karl Meinke. He handled me with care when I set foot on the trail, lifted my spirits when I felt reluctant to climb a cliff, held my hand when I slipped from a ridge and challenged me when he felt that it was time to go the distance. I owe my deepest gratitude to you GURU for all your hard work and patience during all these years.

I thank the Higher Education Commission (HEC) of Pakistan for giving me the scholarship for this PhD study and the School of Computer Science & Communication (CSC), KTH which financed me after the scholarship period finished.

I thankfully acknowledge the role of my departmental chairman, Prof. Muhammad Afzal Bhatti at Quaid i Azam University, Islamabad, Pakistan in ensuring that a leave is sanctioned in my favour as a special case to pursue this PhD study.

I would like to thank our former heads of TCS department Stefan Arnborg and Johan Hästad who were very kind whenever I turned up to them for help or advice. Stefan also reviewed the drafts of my Licentiate and PhD theses for which he is duly thanked.

I thank all the graduate students in the TCS department for providing me with a friendly work environment and giving useful comments whenever I sought from any of them. A bundle of thanks to my office mates Andreas Lundblad and Niu Fei with whom I shared several light moments in the office. Their company made this journey a lot more easier.

I am obliged to thank Prof. Bengt Jonsson from Uppsala University who was the examiner of my Licentiate thesis and gave me useful feedback for future research.

Finally, I would like to mention my beloved parents, wife, brother and sisters who have very patiently supported me during my stay in Sweden and eagerly await my arrival back home. I thank you all from the core of my heart because your prayers and well wishes made it possible for me.
Contents

I Introduction and Literature Survey 1

1 An Overview of this Thesis 3
   1.1 LBTest as an Implementation of LBT 4
   1.2 Organization of the Thesis 4
   1.3 Overview of Research Papers 5

2 Introduction and Background to Software Testing 7
   2.1 Introduction 7
   2.2 Specification Based Testing 8
   2.3 Black Box and White Box Testing 9
      Black Box Testing 9
      Random Testing 9
      White Box Testing 10
   2.4 Conformance Testing 10
   2.5 Model Based Testing 11
   2.6 Learning Based Testing 12
   2.7 Inductive Testing 13
   2.8 Static Checking 13
      Manual Static Checking 14
      Automated Static Checking 14
   2.9 Formal Verification 15
      CEGAR: Inductive Verification 15

3 Principles of Finite Automata 17
   3.1 State Machines and Formal Languages 17
      Definition (Deterministic Finite Automata) 17
      Definition (Moore Machine) 18
      Definition (Mealy Machine) 19
      Definition (Kripke Structure) 19
## CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Reactive Systems</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>A Cruise Controller</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>A 3-Floor Elevator</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>Automaton Learning Theory</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Strings and Languages</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>Automata Learning</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>L* Algorithm</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>ID Algorithm</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>IDS Algorithm</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>L* Mealy Algorithm</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>Other Algorithms</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>RPNI2 Algorithm</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>CGE Algorithm</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>IKL Algorithm</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>Basic Complexity Results</td>
<td>37</td>
</tr>
<tr>
<td>5</td>
<td>Model Checking</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Introduction</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Basic Ideas</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Temporal Logic</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>Linear Temporal Logic (LTL)</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>Syntax of PLTL</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>Semantics of PLTL</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>Model Checking</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>NuSMV Model Checker</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>SMV Language</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>Expressing LTL in NuSMV</td>
<td>43</td>
</tr>
<tr>
<td>6</td>
<td>Requirements Languages</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>What is a requirement language?</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>Types of requirement languages</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>Propositional linear temporal logic for finite data types</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>Definition (ADT)</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>Example (ADT)</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>Definition PLTL(Σ)</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Semantics of PLTL(Σ)</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Finite Data Type Modeling in LBTest</td>
<td>51</td>
</tr>
<tr>
<td>7</td>
<td>Coverage Models in Software Testing</td>
<td>55</td>
</tr>
</tbody>
</table>
## CONTENTS

7.1 What is coverage in SW Testing? ........................................... 55
7.2 Coverage in White-box Testing ............................................. 56
7.3 Coverage in Requirements-based Black-box Testing ................. 56
7.4 Coverage in LBTest .......................................................... 57
    Analysis of LBTest Results ............................................. 58
    State Coverage ....................................................... 58
    Transition Coverage ............................................... 59
    Edge-Pair Coverage ............................................... 59
    Prime Path Coverage ............................................... 59
7.5 Prospects of Requirements Coverage .................................... 59

8 Software Testing Tools .......................................................... 61
8.1 Testing Tool Survey ......................................................... 61
    LTG/B & LTG/UML ................................................... 62
    Spec Explorer ......................................................... 62
    STG ................................................................. 63
    UPPAAL-TRON ....................................................... 63
    JUMBL ............................................................. 64
    LBTest ............................................................. 64
8.2 Tool Comparison ............................................................ 66

9 Conclusions and Future Work .................................................. 67
9.1 Summary ..................................................................... 67
9.2 Contributions of the thesis .................................................. 68
9.3 Author’s personal contribution ............................................. 68
    Paper 1 .................................................................. 69
    Paper 2 .................................................................. 69
    Paper 3 .................................................................. 69
    Paper 4 .................................................................. 69
    Other Papers ................................................................ 69
9.4 Future Work ................................................................. 70

Bibliography ........................................................................ 73

II Included Papers .................................................................. 81

A Paper 1 (Incremental Learning-based Testing for Reactive Sys-
               tems) ................................................................ 83

B Paper 2 (An Efficient Model Inference Algorithm for Learning-
               based Testing of Reactive Systems) ...................... 103

C Paper 3 (LBTest: A Learning-based Testing Tool for Reactive
               Systems) ........................................................... 133
D  Paper 4 (Case Studies in Learning-based Testing) 151
Part I

Introduction and Literature Survey
Chapter 1

An Overview of this Thesis

In this thesis, we investigate the feasibility of learning-based testing (LBT) ([MS11], [MNS12]) for black-box requirements testing of reactive systems. The basic idea of LBT is to automatically generate a large number of test cases by combining an incremental automata learning algorithm or a model inference algorithm with a model checking algorithm. The three steps fundamental to LBT are: i) automated test case generation (ATCG), ii) execution of tests generated in step (i) and iii) assigning a verdict on a test outcome. The system under test (SUT) is integrated with both these algorithms in an iterative feedback loop. This optimizes test case generation based on previously observed outcomes of test cases. A flow diagram which illustrates this principle is shown in Figure 1.1. This thesis will address the following questions about LBT:

- Is LBT practically feasible for reactive and embedded systems for which soft-
ware quality requirements are usually high?

• Which efficient algorithms are available for LBT?
• How effective is LBT in finding errors?
• How quickly does LBT discover errors?
• How scalable is LBT to industrial case studies?

1.1 LBTest as an Implementation of LBT

The whole LBT framework has been realized inside a tool called LBTest as part of research on this thesis. LBTest is a testing tool that automatically generates a large number of test cases to test both safety and and liveness properties of reactive systems. It also has the features of automatic verdict assignment, result reporting and visualization of generated automata and verdicts on screen. More details about LBTest will appear in Part I, Chapter 8 and Part II, Paper 3.

A PhD thesis progresses from an abstract idea to something promising and unique after several years of work. In the following paragraphs a brief overview of what happened during this PhD study and what the reader of this thesis can expect to see in the appended research papers. This thesis consists of two parts. The first part consists of nine chapters and the second part consists of four chapters. The first part provides the necessary background and literature survey to understand the second part which contains four research papers produced as a result of this thesis.

1.2 Organization of the Thesis

The main subject of this thesis concerns software testing for reactive systems. Therefore, Chapter 2 reviews different existing testing techniques used by software testers for different testing needs and an overview of reactive systems and their underlying mathematical principles are outlined in Chapter 3.

This thesis makes use of automata learning to infer models of systems which are to be tested, a brief overview of learning theory with focus on automata learning algorithms is provided in Chapter 4. The pros and cons of these algorithms for the purpose of testing is also analyzed in this chapter.

Chapter 5 provides a basic background to model checking which is essential to understand how test cases are automatically generated from the models inferred by learning algorithms. This chapter also introduces temporal logics which are used as a requirements language to describe properties of reactive systems.

In Chapter 6, a review of several requirements languages especially in the context of formal methods is given. An introduction to a new requirements language PLTL(Σ) is also given which extends the temporal logics introduced in Chapter
5 with finite data types. This language is the requirements language used by the LBTest tool.

Chapter 7, gives an overview of various coverage models used in different areas of software testing to quantify the testing effort. A coverage study done with LBTest tool is also provided in this chapter. The results obtained give some interesting insight into existing coverage models in the literature.

In Chapter 8, we give a review of several testing tools. These are somewhat different in nature from LBTest but still share some similarity of goal (testing functional properties), method (use behaviour models) and result (assign verdicts to test cases) for black-box testing of SUTs with LBTest.

1.3 Overview of Research Papers

In the second part of this thesis research papers produced as a result of this thesis are appended.

Paper 1 presents the underlying architecture and a brief introduction to the learning algorithm in LBTest, which is the IKL algorithm. This algorithm along with some preliminary results for learning-based testing are presented in this paper. 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.

Paper 2 gives a detailed description and a proof of correctness of the IKL learning algorithm including a minimization algorithm which is used when the reachable product automata generated by the IKL algorithm is highly non-minimal. This paper also gives some heuristics for convergence of the learned SUT.

Paper 3 describes the LBTest tool along with its main workflows with the help of a pedagogical case study. This aims to enable the user to set up the tool for testing and familiarize himself / herself with it in a readily comprehensible manner.

In Paper 4, two industrial case studies are described which were tested with the LBTest tool. The results obtained from these case studies show that learning-based testing can indeed be scaled to industrial strength applications. The results obtained from a coverage study performed with the LBTest tool are also analyzed and described in this paper.

The author’s personal contribution to each of these four papers is described in Part I, Section 9.3.
Chapter 2

Introduction and Background to Software Testing

2.1 Introduction

The purpose of software testing is to strengthen belief that a software program works as desired but most importantly 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 [Som10] 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 [Boe78]. 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 [DDH72]).

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 4) to iteratively learn the system under test (SUT). The iteratively learned model is then model checked (described in Chapter 5) against a specific requirement formula expressed in temporal logic (described in Section 5.3). 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 2.2) using a requirement formula and the outcome of the test case. Before we consider LBT in more detail (in Section 2.6), it is appropriate to begin by reviewing some different but related testing techniques.

### 2.2 Specification Based Testing

The development of any engineering system should begin 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 stakeholders’ 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 in software engineering may vary depending upon context. For example it is common to hear terms like requirements specification, design specification and module specification in software engineering. These terms are used during different phases of software development and have a different meaning depending upon context. Formal descriptions are desirable for automating testing and verification.

A specification can be described in a formal or an informal style. In the informal style, the description is in natural language and can use visual aids like diagrams, tables and other visual notations to enhance 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 usually 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 judged to be passed 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 the case of informal specifications because automating the testing process from them is non-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 pass verdict if the expected set of values contains the observed values, otherwise the verdict is a fail.

2.3 Black Box and White 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. 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 [Jor08]), specification-based testing (see Section 2.2) and random testing (see Section 2.3).

The equivalence partitioning approach for black-box testing mentioned above divides input data into a finite partition of equivalence classes. Test cases from these classes are derived in such a way that each class is covered by at least one test case. This concept will be used later in this thesis in Section 6.3.

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 [Ham02] as, “Random testing, of course, is the most used and least useful method.”

The question however is why a random technique should be used instead of a systematic testing technique? In [Ham02] 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 over several trials and experiments only random variations can be averaged out and refined to yield a better result. This may not be the case for systematic variations 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). Therefore random testing can be used as an effective tool when 
a large number of test cases has to be generated and for benchmarking of other 
testing techniques against random testing because of its statistical nature.

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 
[AO08]). 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 \( \leq 1 \) and length \( \leq 2 \) 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 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 a 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.

2.4 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. The formal specification in 
this case is a global automaton model of the SUT that captures its functionality. 
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 [Tre96]. 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(model). For example when the specification allows two possible outputs for a particular input then the corresponding \textit{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 exercise at least as many inputs as the specification and at most gives as many outputs (can be less) as the specification.

The methodology is that 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 \textit{sound} if every implementation that conforms to the specification also passes the test suite. Conversely a test suite is \textit{complete} if every non-conforming implementation fails the test suite. For practical reasons it is impossible to achieve \textit{completeness} because every non-trivial system will require an infinite number of test cases to be executed before reaching completeness. Therefore an incomplete test suite should at least be \textit{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.

\section{Model Based Testing}

\textit{Model-based testing (MBT)} (see [UL07]) involves the use of a \textit{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 derived from this abstract model are also abstract and are part of an \textit{abstract test suite (ATS)}. The ATS cannot however be executed on an SUT directly, rather it has to be converted into an \textit{executable test suite (ETS)} by some means for execution on a concrete SUT. Test cases are derived from models in the 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 by using e.g graph theory.

Model-based testing can be carried out either \textit{online} or \textit{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 proving (see [HNS97]), constraint logic programming and symbolic execution (see [DO91]) and more recently model checking (see [FWA09]).

2.6 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 validity checked against the formal specification for SUT and a counterexample is returned in case of a failure of validity. 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 [Mei04] and [MN10] and to reactive SUTs in [MS11].

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 [PVY99], [GPY06] and [RSM08]. Reactive systems, learning and model checking will be reviewed in Chapter 3, Chapter 4 and Chapter 5 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 2.9). 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.

2.7 Inductive Testing

Inductive testing is a heuristic approach to black-box testing. Like LBT, the heuristic idea of inductive testing is also to learn a black box system using tests as queries. However unlike LBT, 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 [WBDP10] 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 [WBDP10] 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.

2.8 Static Checking

We will contrast specification based testing with other methods for software quality assurance such as formal methods based analysis 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 complementary 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
CHAPTER 2. INTRODUCTION AND BACKGROUND TO SOFTWARE TESTING

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 (see [Som10]).

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 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 [WCC+95]) tool uses directed graphs and regular algebra to prove
that software being analyzed meets its mathematical specification). ESC/Java (see
[FLL+02]) 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 theorem proving and a simplified semantic model of Java code. Extended
static checking 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.

2.9 Formal Verification

The type of static analysis in which general properties of a program are proven
mathematically is called formal verification. Its use is increasing 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 fi-
nite 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 5.

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 lim-
itations such as the 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 reduced in severity by the introduction of binary decision diagrams
(BDDs) see e.g. [BCM+90]. This approach was used in a well known model checker
developed around that time called NuSMV (see e.g [NuS]). But the state space ex-
losion 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 abstrac-
tion 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 2.6, an abstraction can be built automatic-
cally 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 CE-GAR is on formal verification (or at least static checking) and complete learning rather than testing and incremental learning.
Chapter 3

Principles of Finite Automata

In this chapter we introduce the classes of objects to be learned in LBT which are finite automata. There is an extensive literature on this model of computation such as [HMU06]. Therefore we need to review only the most essential concepts here. Finite automata can be used as models of reactive systems (will be described in Section 3.2). We can test any SUT by using LBT which conforms to these models.

3.1 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 $\Sigma$ be any set of symbols (alphabet) then $\Sigma^*$ denotes the set of all finite strings over $\Sigma$ including the empty string $\epsilon$. The length of any string $\alpha \in \Sigma^*$ is denoted by $|\alpha|$ and $|\epsilon| = 0$. For any two strings $\alpha_1, \alpha_2 \in \Sigma^*$ their concatenation $\alpha_1 \alpha_2$ denotes their concatenation.

Definition (Deterministic Finite Automata).

A deterministic finite automata (DFA) $A$ is a quintuple $(Q, \Sigma, \delta, q_0, F)$ where:

- $Q$ is a finite set of states,
- $\Sigma$ is a finite set of input symbols,
- $\delta : Q \times \Sigma \rightarrow Q$ 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 $\delta$ can be inductively lifted to $\delta^* : Q \times \Sigma^* \rightarrow Q$, where $\delta(q, \epsilon) = q$ and $\delta^*(q, \sigma_1, ..., \sigma_n) = \delta(\delta^*(q, \sigma_1, ..., \sigma_{n-1}), \sigma_n)$. A string $\beta$ of the form $\sigma_1, ..., \sigma_n$ is accepted by $A$ iff
The language accepted by $A$, denoted by $\mathcal{L}(A)$, consists of all strings $\sigma_1...\sigma_n \in \Sigma^*$ which are accepted by $A$ i.e $\delta^*(q_0, \sigma_1...\sigma_n) \in F$. The set $Q$ of states may or may not contain a dead state $d_0$. The dead state $d_0 \notin F$ and $\delta(d_0, \sigma) = d_0$ for all $\sigma \in \Sigma$.

For any given DFA $A$ there exists a minimum state DFA $A'$ such that $\mathcal{L}(A) = \mathcal{L}(A')$ 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 $\delta$ can be inductively lifted to $\delta^*$ as in the definition of DFA earlier in this section.
Definition (Mealy Machine).

A Mealy machine $M_{ly}$ is a six-tuple $M_{ly} = (Q, \Sigma, \Omega, q_0, \delta, \lambda)$ 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,
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function,
- $\lambda : Q \times \Sigma \rightarrow \Omega$ is the output function,
- $q_0 \in Q$ is the initial state.

Then $\Sigma^*$ and $\Omega^*$ represent the set of all finite sequences of inputs and outputs over $\Sigma$ and $\Omega$ respectively.

Note that Moore and Mealy machines can be translated into one another albeit with a possible increase in state space size. Generally, Mealy machine models are more compact. A generalization of the Moore machine is defined in the following section.

Definition (Kripke Structure).

A (non-deterministic) Kripke structure $K$ over a set $AP$ of atomic propositions is a five-tuple $K = (Q, \Sigma, \delta, q_0, \lambda)$ 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,
- $\lambda : Q \rightarrow 2^{AP}$ is a labelling function for states.

We say that $K$ is deterministic if $\delta$ is a function $\delta : Q \rightarrow Q$. 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 $\lambda$.

From a learning perspective we want to work with Kripke structures as Moore machines with states labelled by Boolean vectors. So 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,
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function,
- $q_0 \in Q$ is the initial state,
- $\lambda : Q \rightarrow \mathbb{B}^k$ 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 3.1 we let $\delta^* : Q \times \Sigma^* \rightarrow Q$ 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 $\lambda^* : \Sigma^* \rightarrow \mathbb{B}^k$ denote iterated output function $\lambda^*(\sigma_1, ..., \sigma_n) = \lambda(\delta^*(q_0, \sigma_1, ..., \sigma_n))$.

3.2 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 of as a reactive system. A distinctive feature of such systems is that their behaviour can be modeled as automata (such as the algebraic structures discussed in Section 3.1). Also their behaviour can be specified using temporal logics (temporal logics will be reviewed in Section 5.3) 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, these are:

1. a simple cruise controller (appears in paper 1, 3 and 4)
2. a 3-floor elevator (appears in paper 1)
A Cruise Controller

A cruise controller (cc) (see e.g. [Gmb07]) 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 3.1. The input set of the cruise controller cc consists of the set \{brake, dec, gas, acc, button\}. The value brake 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 \texttt{mode=00} in the figure represents the manual mode, \texttt{mode=01} represents the cruise mode and \texttt{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 \texttt{speed=00}, \texttt{speed=01} and \texttt{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 \texttt{speed=01} is allowed and in disengaged mode the cc can have speed values of 0 or \texttt{speed=00} representing too low speed or 2 or \texttt{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.

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 3.2. This particular representation is as a hierarchical state chart which can be flattened to a conventional Moore machine. 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 3.1: 5-bit Cruise Controller
Figure 3.2: 3-Floor Elevator
Chapter 4

Automaton 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 $D = d_1, d_2, ..., d_n$ 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 capable of 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 (labels) for data elements $d_i \in D$ and these are used to actually construct approximations $h \in \mathcal{H}$ of $s \in S$ 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 (absent labels) then the problem of inferring the hidden structure from this unlabelled data 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 $s(d_i) - h(d_i)$ or $h(d_i) - s(d_i)$ 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 $h_1, h_2, \ldots$ after each membership query which finitely converge to $s$ i.e. $h_n = s$ for some $n \in \mathbb{N}$. If in addition each hypothesis $h_{i+1}$ is constructed using 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 about $s$ 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:

4.1 Strings and Languages

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

$$L_{\text{reg}} ::= \emptyset \mid \{\epsilon\} \mid \{a\} \mid L_{\text{reg1}} \cup L_{\text{reg2}} \mid L_{\text{reg1}} \cdot L_{\text{reg2}} \mid L_{\text{reg}}^*$$  \hspace{1cm} (4.1)

In other words, the empty language $\emptyset$, the empty string language $\underline{\epsilon}$, the singleton language $\{a\}$ where $a \in \Sigma^*$ are all regular languages. Similarly union “$\cup$” and concatenation “.” of two regular languages $L_{\text{reg1}}$ and $L_{\text{reg2}}$ is also a regular language. The Kleene star construction $L_{\text{reg}}^*$ applied to a regular language also gives a regular language consisting of a concatenation of a finite number of words in $L_{\text{reg}}$.

If $S_1$ and $S_2$ are sets then $S_1 \oplus S_2$ denotes the symmetric difference of $S_1$ and $S_2$ which means those elements that are either in $S_1$ or $S_2$ but not both. The cardinality of the set $S$ is denoted by $|S|$.
4.2 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 $M$. 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 when the Learner asks whether a given string $\alpha \in \Sigma^*$ is in $L(M)$.

- A query to the Oracle is called an equivalence query when the Learner asks whether the approximation or hypothesis $H$ has become equal to $M$ or not. If they are not equal the Oracle will provide a counterexample either from $L(M) \neq L(H)$ or $L(H) \neq L(M)$.

There are many algorithms for learning state machines in the literature including [Gol67], [TB73], [Ang81], [Ang87], [RS93], [Dup96], [PNH98], [Mei10] and [KV94]. Most algorithms are for complete learning of deterministic finite automaton DFA such as [Ang81] and [Ang87]. However, some algorithms are for incremental learning such as [PNH98], [Dup96], [MN12] and some for sequential learning such as [Mei10]. Most of 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 [Gol67] where he showed that a regular language $L(M)$ corresponding to a DFA $M$ can be guessed by a finite number of wrong guesses(hypotheses) about $M$ by using some inference or learning algorithm for DFA $M$. 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 [Ang87], Angluin’s ID algorithm introduced in [Ang81] and the IDS algorithm introduced in [MS10].

4.3 $L^*$ Algorithm

Angluin’s $L^*$ algorithm see [Ang87] 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 $OT$ which is a tuple $OT = (P, S, T)$ for a given alphabet $\Sigma$ 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.

• $T : ((P \cup P.\Sigma) \times S) \rightarrow \{\text{acc, rej, unkn}\}$ is a function or a 2-dimensional table which satisfies the property $ps = p's'$ i.e. $T(p, s) = T(p', s')$ 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 $\{\text{acc, rej, unkn}\}$ 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. If $ps \notin \mathcal{L}(\mathcal{M})$ then it will be rej and if the membership of $ps$ is not known at that particular instance then it will be unkn.

Function $\text{row}(p)$ is a finite function from $S$ to $\{\text{acc, rej, unkn}\}$ for every $p \in (P \cup P.\Sigma)$ and is defined by $\text{row}(p)(s) = T(p, s)$ or more simply $\text{row}(p)$ represents the tuple of entries in the observation table corresponding to the row labelled $p$. All distinct rows of the form $\text{row}(p)$ 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 there are no unkn entries and for each $p_1 \in P.\Sigma$ there exists $p_2 \in P$ such that $\text{row}(p_1) = \text{row}(p_2)$ and $\mathcal{OT}$ is consistent provided that whenever $p_1, p_2 \in P$ such that $\text{row}(p_1) = \text{row}(p_2)$ then for all $\alpha \in \Sigma$, $\text{row}(p_1.\alpha) = \text{row}(p_2.\alpha)$. Replacing unkn entries by acc or rej results in what we term membership or 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 $\Sigma$, with state set $Q$, initial state $q_0 \in Q$, accepting states $F \subseteq Q$ and the transition function $\delta$ by:
4.4. ID ALGORITHM

- \( Q = \{ \text{row}(p) : p \in P \} \),
- \( q_0 = \text{row}(\epsilon) \)
- \( F = \{ \text{row}(p) : p \in P \text{ and } T(p) = \text{acc} \} \),
- \( \delta(\text{row}(p), \alpha) = \text{row}(p \cdot \alpha) \)

The \( L^* \) algorithm maintains the observation table \( OT \) and the sets \( P \) and \( S \) are both initialized to \( \{ \epsilon \} \). Then \( L^* \) will perform membership queries for each \( \alpha \in \Sigma \) and \( \epsilon \) which will result in either an \( \text{acc} \) or \( \text{rej} \) for each query and corresponding fields in each row of \( OT \) are filled with these values. Afterwards \( OT \) 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 \( \text{row}(p_1) = \text{row}(p_2) \) but \( T(p_1 \alpha, s) \neq T(p_2 \alpha, s) \) and adding the new suffix \( \alpha s \) to \( S \) and filling in the \( \text{unkn} \) entries by asking membership queries.

If \( OT \) is not closed then \( L^* \) finds \( p \in P \) and \( \alpha \in \Sigma \) such that \( \text{row}(p \alpha) \neq \text{row}(p \tau) \) for all \( \tau \in P \) 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 \( OT \) has become consistent and closed then the hypothesis \( H \) can be constructed and checked for correctness against the target \( M \) by an equivalence query to the Oracle. If the answer to the equivalence query is \( \text{"yes"} \) then \( L^* \) terminates with a correct hypothesis \( H \) as output. Otherwise the Oracle will provide a counterexample \( \beta \), such that \( \beta \in \mathcal{L}(M) \iff \beta \notin \mathcal{L}(H) \) and \( L^* \) will extend \( OT \) with \( \beta \) and all its prefixes by asking membership queries.

4.4 ID Algorithm

The ID algorithm introduced in [Ang81] is a complete learning algorithm. Unlike \( L^* \) it assumes the availability of a \textit{live complete set} \( P \) of strings for the target DFA \( A \). A state \( q_i \in Q \) is said to be \textit{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 \textit{live} string and a set consisting of at least one such string for each live state of a given DFA is called a \textit{live complete set} and is denoted by \( P \). A state that is not live will be called a \textit{dead state}. A canonical DFA has only one dead state.

The ID algorithm proceeds in the following steps:

1. Initializations: \( i = 0; v_1 = \epsilon; V = \{ \epsilon \}, T = \{ P \cup \{ \text{f}(\alpha, \beta) \} : (\alpha, \beta) \in P \times \Sigma \} \), where \( \text{f} \) is a counter that will count the number of distinguishing strings \( v \). \( V \) is the set of all \textit{distinguishing strings} \( v \). \( P \) is the live complete set, \( f \) is a special concatenation function such that \( f : P' \times \Sigma \rightarrow \Sigma' \), 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 \( A \) 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 \( E_i(\alpha) = \{ v_j | v_j \in V, 0 \leq j \leq i, \alpha v_j \in \mathcal{L}(A) \} \) 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 \( \alpha \in \mathcal{L}(A) \) 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 \( i + 1 \) th 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 \( \upsilon_{i+1} = \sigma \gamma \) 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 \( \alpha \upsilon_{i+1} \in L(A) \). If the answer is “yes” then \( E_{i+1}(\alpha) \) is set to \( E_i(\alpha) \cup \{ \upsilon_{i+1} \} \) 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 \( A \) 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 \( \delta \) of \( \mathcal{H} \) is constructed by adding self loops to all states represented by \( E_m(\alpha) \) if \( E_m(\alpha) = \emptyset \) otherwise \( \delta \) is set as \( \delta(E_m(\alpha), \sigma) = E_m(f(\alpha, \sigma)) \)

7. **ID** outputs the description of hypothesis \( \mathcal{H} \) and stops.

One key feature of the **ID** algorithm is the use of a dead state to represent the unknown part of the system under learning. This feature has been used to devise incremental variants of the **ID** algorithm (see [PNH98], [MS10]). The later will be described in the next section.

### 4.5 **IDS** Algorithm

The **IDS** algorithm introduced in [MS10] 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 rebuilds the hypothesis incrementally after each external query. These external queries are retrieved from a file \( S \) or an interface which gives
4.6. $L^*$ MEALY ALGORITHM

a sequential order to the stream of examples. It assumes the availability of an adequate 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 incrementally constructs a family of hypothesis $\mathcal{H}_1, \mathcal{H}_2, \ldots$ 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 $\Sigma$, $\alpha \in \Sigma$ 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 $\alpha$), if $\alpha$ 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 $\alpha$. The steps for the IDS algorithm are given below. The sets $\mathcal{P}, \mathcal{P}', \mathcal{T}, \mathcal{T}', \mathcal{V}$ 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 4.4:

1. Initializations: $i = 0; k = 0; m = 0; \nu_0 = \epsilon, V = \{\nu_0\}$
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 4.4 above.
5. Construct the representation of hypothesis automata $\mathcal{H}_m$ as described in point 6 section 4.4 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 $k = k + 1, m = m + 1, P_k = Pref(\alpha) \cup P_{k-1}, P'_k = P_k \cup \{d_0\}, T_k = T_{k-1} \cup Pref(\alpha) \cup \{f(\alpha, \beta)|\alpha, \beta \in P_k \times \Sigma\}, T'_k = T_{k-1} \cup \{d_0\}$ 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: $E_i(\alpha) = \{\nu_j|0 \leq j \leq i, \alpha \nu_j \in \mathcal{L}(\mathcal{A})\}$, go to step 4.

4.6 $L^*$ Mealy Algorithm

Another complete learning algorithm was given by [Nie03] 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 as $L^*$. It constructs a
hypothesis when suitable information has been accumulated in the observation table $OT$ 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 $Mly = (\Omega_T, Q_T, q_0^T, \delta_T, \lambda_T)$. The SUT will not provide accept/reject responses instead it will provide a response $\tilde{\lambda}$ from a set $\Omega_T$ of output symbols. The observation table $OT$ entries will consist of strings from $\tilde{\Omega}$ and an unknown value unk. The function $T$ maps row and column labels to strings of output symbols and can be defined as $T : (P \cup P.\Sigma) \times S \rightarrow \tilde{\Omega} \cup \{unk\}$. $T(p, \sigma)$ is changed to $\tilde{\lambda}$ by applying the iterative output function $\lambda_T(\delta_T(q_0^T, ps), \sigma) = \tilde{\lambda}$ where $p \in P$, $\sigma \in S$, $\sigma \in \Sigma$ and $\sigma \in \Omega_T$. 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 a hypothesis Mealy automaton $Mly_H$ can be constructed by:

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

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 $Mly_H$ and $Mly$ is returned otherwise, in case of a positive answer from the equivalence oracle the $L^*$ Mealy algorithm will terminate.

### 4.7 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 4.2. 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 learned or tested.

3. Membership queries can be expensive in terms of execution time of the learning algorithm as observed in [BJ08].

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 membership queries for structural purposes (congruence closure) should be minimal.

For these reasons in this thesis 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 [Dup96]) and the second was given by Meinke (see [Mei10]). Both these algorithms are sequential in nature.

**RPNI2 Algorithm**

Dupont’s RPNI2 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 [Onc91]. The RPNI algorithm is a passive learning algorithm which requires the positive and negative information to be given as a whole. This renders it irrelevant for learning based testing when new learning data becomes available as the whole process has to be restarted. The RPNI2 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 RPNI2 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 membership queries to yield a hypothesis.

**IKL Algorithm**

The IKL algorithm is a multi-bit incremental extension of the Angluin’s ID algorithm. A detailed version of this algorithm appears in Part II Paper 1 and Paper 2 of this thesis. Here we discuss only the salient features of this algorithm. The IKL extends simple DFA learning 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 \( \mathcal{A}_1, \ldots, \mathcal{A}_k \) with 1-bit output. These component Kripke structures can be learnt by any regular inference algorithm. The inferred Kripke structures \( \mathcal{B}_1, \ldots, \mathcal{B}_k \) 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 \( E_{i_1}^1, \ldots, E_{i_k}^k \) 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 \( E_{i_j}^j \) a set of distinguishing strings \( V_j \) is iteratively generated and equivalence classes in \( E_{i_j}^j \) 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 \( E_{i_j}^j \) which is not yet a congruence. On the other hand if \( E_{i_j}^j \) is already a congruence then it is not refined further. This helps in minimizing the membership queries of the IKL algorithm which are not useful from the perspective of testing as these usually do not make interesting requirements based test cases.

Figure 4.1 illustrates the working of the IKL algorithm with the help of an example consisting of a 3-bit Kripke structure. The 3-bit Kripke structure on the top of the figure is sliced into three 1-bit automata as shown in the second layer. Each of these 1-bit automata are then learned by the IKL algorithm to yield three 1-bit hypothesis automata as shown in the third layer of Figure 4.1. The reachable product of these three 1-bit automata is then computed by the IKL algorithm which is shown at the bottom of the figure. It can be seen that this product automata is behaviourally equivalent to the target automata on the top of Figure 4.1. Notice that the product automata in this case is already minimal therefore the minimization step is not shown in Figure 4.1 which otherwise would have been computed by the IKL algorithm.

A summary which compares the features of the algorithms discussed so far is given in Table 4.1.
4.7. OTHER ALGORITHMS

Figure 4.1: Illustration of IKL Algorithm
### Table 4.1: Learning algorithm comparison

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Type</th>
<th>Complete</th>
<th>Consistency</th>
<th>Closure</th>
<th>Consensus</th>
<th>Membership</th>
<th>Learning Type</th>
<th>Average</th>
<th>falsely</th>
</tr>
</thead>
<tbody>
<tr>
<td>L*</td>
<td>Mealy</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>None</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>IDS</td>
<td>Mealy</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>None</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>RPNI</td>
<td>Moore</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>None</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>CGE</td>
<td>Mealy</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>None</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>IKL</td>
<td>Moore</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>None</td>
<td>Yes</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Algorithm**: L*, IDS, RPNI, CGE, IKL
- **Type**: Mealy, Moore
- **Complete**: Yes, No
- **Consistency**: Yes, No
- **Closure**: Yes, No
4.8 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 4.2. 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 $RPNI2$, $S_p \subseteq \mathcal{L}(A)$ is a positive sample and $|S_p|$ represents its size similarly $S_n \subseteq \mathcal{L}'(A)$ represents a negative sample and $|S_n|$ represents its size. In case of $CGE$, $n$ represents the longest acyclic path in the automaton $A$.

<table>
<thead>
<tr>
<th>No</th>
<th>Algorithm</th>
<th>Time Complexity $O(\text{Queries})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$L^*$</td>
<td>$O(</td>
</tr>
<tr>
<td>2</td>
<td>$ID$</td>
<td>$O(</td>
</tr>
<tr>
<td>3</td>
<td>$IDS$</td>
<td>$O(</td>
</tr>
<tr>
<td>4</td>
<td>$IKL$</td>
<td>$O(</td>
</tr>
<tr>
<td>5</td>
<td>$L^* Mealy$</td>
<td>$O(\max(N,</td>
</tr>
<tr>
<td>6</td>
<td>$RPNI2$</td>
<td>$O(</td>
</tr>
<tr>
<td>7</td>
<td>$CGE$</td>
<td>$O(</td>
</tr>
</tbody>
</table>

Table 4.2: Learning algorithm time complexities

In a testing context, each query should ideally be an interesting requirements based test case but in practical situations this is not possible due to the reasons described at the beginning of Section 4.7. In papers 1 and 2 we will investigate the $IKL$ algorithm as an efficient learning algorithm for LBT.
Chapter 5

Model Checking

5.1 Introduction

Although we will make use of model checking as a black-box tool, 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 some essential concepts in the field of model checking.

5.2 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 $\phi$ that describes the behaviour of the system. The system is represented as some kind of automaton usually a Kripke structure also called the model of the system and denoted $M$. Usually the formula $\phi$ is taken from a temporal logic such as LTL, CTL or CTL*. We will explain more about these logics in Section 5.3. 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 $\phi$ satisfies the model $M$ then the model checker will report the specification to be true. On the other hand if the specification $\phi$ is violated by the model $M$ 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.
5.3 Temporal Logic

Temporal logics constitute systems of rules and notations pertaining to the representation 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 [Pnu77]. Other categories of temporal logics include Computational Tree Logic (CTL) also called Branching Time Logic introduced by [CE82], CTL* given by [EH82], Hennessy & Milner Logic (HML) given by [HMS85] and Modal µ-calculus given by [Koz83] to name a few.

Linear Temporal Logic (LTL)

Linear time temporal logic or linear temporal logic (LTL) is the most commonly used family of logics 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 $\pi := (q_0, q_1, \ldots)$ in a deterministic Kripke structure $K$ corresponding to an infinite word $w = \sigma_0, \sigma_1, \ldots \in \Sigma^\omega$ 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 PLTL

Definition.

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

$$\phi ::= \bot \mid T \mid p \in AP \mid \neg \phi \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid \phi_1 \rightarrow \phi_2 \mid \bigcirc \phi \mid \Diamond \phi \mid \Box \phi \mid \phi_1 \cup \phi_2 \quad (5.1)$$

The symbols $\bot, T, \neg, \land, \lor$ and $\rightarrow$ are the usual Boolean connectives which have the same meaning in PLTL as in propositional logic. The operators $\bigcirc, \Diamond, \Box$ and $\cup$ are the temporal connectives. Here $\bigcirc \phi$ means that $\phi$ is true in the next state, $\Diamond \phi$ means that $\phi$ is true in some future state, $\Box \phi$ means that $\phi$ is true in all future states including the current state and $\phi_1 \cup \phi_2$ is the binary PLTL operator which
5.4. MODEL CHECKING

means that \( \phi_1 \) will remain true until the state where \( \phi_2 \) becomes true is reached. We can also mention two more PLTL operators \( W \) and \( R \) which stand for Weak Until and Release respectively. The precise semantics of the PLTL formulas above is given in next section.

Semantics of PLTL

The meaning of the temporal operators of PLTL is made more precise by defining a formal semantics for the language. This is conventionally defined in terms of a satisfaction relation \( \models \).

Let \( K \) be a deterministic Kripke structure which models our system and \( \phi \in PLTL \) be a property we want to investigate. If \( \phi \) is satisfied by the path \( \pi \) 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 \( \phi \) is not satisfied by the path \( \pi \) 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 \( K = (Q,\Sigma,q_0,\lambda) \), 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 PLTL formulas is inductively defined on the structure of \( \phi \) as follows:

\[
\begin{align*}
K, \pi \not\models & \bot \quad (5.2) \\
K, \pi \models & \top \quad (5.3) \\
K, \pi \models & p \iff p \in \delta^*(q_0, \sigma_0, \ldots, \sigma_i) \quad (5.4) \\
K, \pi \models & \neg \phi \iff K, \pi \not\models \phi \quad (5.5) \\
K, \pi \models & \phi_1 \land \phi_2 \iff K, \pi \models \phi_1 \land K, \pi \models \phi_2 \quad (5.6) \\
K, \pi \models & \phi_1 \lor \phi_2 \iff K, \pi \models \phi_1 \lor K, \pi \models \phi_2 \quad (5.7) \\
K, \pi \models & \phi_1 \rightarrow \phi_2 \iff K, \pi \not\models \phi_1 \lor K, \pi \models \phi_2 \quad (5.8) \\
K, \pi \models & \Box \phi \iff K, \pi^1 \models \phi \quad (5.9) \\
K, \pi \models & \Diamond \phi \iff \exists i \in \mathbb{N} : K, \pi^i \models \phi \quad (5.10) \\
K, \pi \models & \forall \phi \iff \forall i \in \mathbb{N} : K, \pi^i \models \phi \quad (5.11) \\
K, \pi \models & \phi_1 \cup \phi_2 \iff \exists j \in \mathbb{N} : K, \pi^j \models \phi_2 \land \forall 0 \leq i < j K, \pi^i \models \phi_1 \quad (5.12)
\end{align*}
\]

5.4 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 [LP85], symbolic model checking see [McM93] and bounded model checking see [BCCZ99]. These algorithms can verify correctness properties expressed in several different kinds of temporal logics including LTL discussed in Section 5.3.

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 \( \phi \) 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 \( \phi \). 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 [Hol97]).

Symbolic model checking uses binary decision diagrams (BDD) see [McM93] 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 can adversely impact 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 upper bound. As long as the boundary is not very big this approach is very efficient. The NuSMV model checker see [NuS] 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 supports 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 the Java language’s string manipulation features.
5.5 NuSMV Model Checker

The NuSMV [NuS] 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 [McM93] as well as propositional satisfiability (SAT) based model checking see [BCCZ99]. 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 5.1. 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 3.2 are given below:

- **When the vehicle is going uphill in cruise mode with cruising speed its cruising speed is maintained.**
  
  - $G(\text{mode} = 01 \& \text{speed} = 01 \& \text{input} = \text{dec} \rightarrow X(\text{speed} = 01) )$

- **When the vehicle is in cruise mode and gas pedal is pressed then it will be disengaged.**
  
  - $G(\text{mode} = 01 \& \text{input} = \text{gas} \rightarrow X(\text{mode} = 10) )$
CHAPTER 5. MODEL CHECKING

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 5.1: SMV code for a simple cruise controller

These two are the safety properties for the cruise controller described in Section 3.2. 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 ($mode = 01$).
Chapter 6

Requirements Languages

"A problem well stated is a problem half solved." Charles F. Kettering

In the previous chapter we have reviewed the basic concepts of model checking, including temporal logic models of requirements. In this chapter we will consider how LTL requirements can be adapted to practical requirements based software testing. We also review more general and related form of requirements languages.

6.1 What is a requirement language?

When Charles F. Kettering said the above words he was precisely describing the problem of poor versus good requirements because any engineering activity, including software engineering, starts with a description of a problem. To be able to successfully solve a problem directly depends upon how well the problem is formulated. The format to describe a problem may vary depending upon the intended audience. This format in which a problem is described is called a requirements language. In other words a requirements language is a means to describe what a system does rather than how it is done.

6.2 Types of requirement languages

Natural language can be used as the simplest type of requirements language by providing an informal description of the problem in it. Using natural language as a requirements language has the advantage of a high level and succinct description of requirements which can be easily comprehended by the stake holders of the system. On the other hand natural language requirements can not be easily automated and they can sometimes be ambiguous.

Precision of requirements increases when informal problem descriptions are described in a more precise way following some formal rules giving rise to a formal requirement model or specification. When the requirements language has a formal
syntax but an informal semantics, this gives rise to a semi-formal requirements language. On the other hand, if both the syntax and semantics are formal then the requirements language is called formal. Formal requirements languages can also be classified on the basis of whether they allow a logic representation or a graphical representation to specify requirements.

The history of formal requirements languages can perhaps be traced as far back as the Software Engineering Crisis of the 1960s. It was around this time that the cost of manufacturing computer hardware began to drop rapidly while the cost of software to be run on these machines began to rise steeply because of the prevalent software development practice. Around that time R. W Floyd in his famous paper [Flo67] described how correctness problem in software can be addressed by using program properties described in first order logic. The work of Floyd was further extended by C. A. R Hoare when he proposed a formal system of rules to argue about the correctness of programs referred to as Hoare Logic or Floyd-Hoare logic (see [Hoa69]).

Amir Pnueli in his work in [Pnu77] introduced another very useful class of formal requirements languages which are modal temporal logics and enable the user to specify temporal properties (both past and present) of a system. In [Pnu77] this kind of logic was proposed for formal verification of systems.

Another important and useful type of formal specification language is called an algebraic specification language. It is based on algebra rather than logic as its underlying formalism. It is used to define the system as a heterogenous or a many-sorted algebra. A heterogenous algebra consists of a collection of different sets with several operations defined over them. The collection of sets, constants and the related operations are specified in the signature of the specification. These constants and operations are then axiomatised using equations. This axiomatisation is also called equational specification. Models of an equational specification capture possible semantics for the specification. On the basis of axiomatisation further data properties can be inferred using equational logic.

Several researchers during the 70s and 80s worked to explore and extend the field of many-sorted algebras. In [Gri78] for example a survey of the subject during that period is provided. The use of equations to model data types along with many-sorted algebras was first used in [LZ75]. This approach was further developed in several later research papers influencing this field e.g [Gut75], [GH78], [GTW78] and [Gog89].

6.3 Propositional linear temporal logic for finite data types

Recall from Section 5.3 that PLTL because of its path or execution sequence oriented nature seems to be a very good option for testing purposes. However, PLTL supports only Boolean data types and connectives which makes it more suitable for testing control properties rather than functional properties. This characteristic of PLTL is an impediment to test many practical systems. In this section we in-
6.3. PROPOSITIONAL LINEAR TEMPORAL LOGIC FOR FINITE DATA TYPES

Introduce the language PLTL(\(\Sigma\)) *propositional linear temporal logic* over finite data types which is supported by LBTest. This extension with finite data types is needed for two reasons:

- to bridge the gap between fixed bit vector encodings possible in Boolean data types and general infinite data types like integers, floats, strings etc.
- to provide formal logical support for partition testing (described in Section 2.3) of infinite data types.

It is necessary to describe the data type model supported by LBTest before we give a formal syntax and semantics of PLTL(\(\Sigma\)). This model is based on *abstract data types* involving heterogenous or many-sorted algebras (see [LEW96]). To keep the model checking problem decidable for this extension of PLTL only finite data types are considered for use in LBTest. This finite data type model can be efficiently used for infinite data types in the presence of an externally defined partition relation on infinite data types. The purpose of such a partition relation is to abstract infinite data types into finite ones.

**Definition (ADT)**

1. A *finite data type signature* \(\Sigma\) consists of a finite set \(S\) of *sorts* or types, and for each sort \(s \in S\), a finite set \(\Sigma_s\) of *constant symbols* all of the same type \(s\).

2. If \(\Sigma\) is a finite \(S\)-sorted data type signature then a *concrete* \(\Sigma\) *data type* \(A\) consists of: (a) a finite set \(A_s\) for each sort \(s \in S\), and (b) for each sort \(s \in S\) and each symbolic constant \(c \in \Sigma_s\) a concrete value \(c_A \in A_s\).

Note that constructors, data operations and selection functions are absent in the above definition of a *finite data type* and hence these are not supported in LBTest. This gives a very simple data model which at present is supported by LBTest for user requirements modeling.

**Example (ADT)**

In this example we would like to represent continents of the world as an abstract data type with their concrete representation consisting of three binary bits for each continent. Therefore in this case the sort is, \(s = \text{continent}\) and \(S = \{\text{continent}\}\) is the set of sorts. The set of constant symbols is,

\[\Sigma_a = \{\text{Africa}, \text{Antarctica}, \text{Asia}, \text{Australia}, \text{Europe}, \text{N. America}, \text{S. America}\}\]
The concrete data type $A$ consisting of three bit binary representation of each continent is given by $A = \{000, 001, 010, 011, 100, 101, 110\}$ where $\text{Africa}_A = 000$, ..., $\text{South America}_A = 110$. We can see that each $c \in \Sigma_s$ has a corresponding member $c_A \in A_s$.

If we reverse the order of bits in the concrete $\Sigma$ data type it will give rise to an entirely different concrete data type $B$ which can be a cause of communication problems between different realizations or implementations of $\Sigma$. This is because the sender and receiver will be unable to recognize the most significant and the least significant bits in the bit-vector in the absence of a protocol. LBTest requires such a protocol to be implemented in a wrapper program on the side of SUT (which acts as a server) during testing and LBTest itself acts as a client program.

Definition PLTL($\Sigma$)

Let $S$ be a finite set of sorts containing a distinguished input type $in \in S$, and let $\Sigma$ be a finite data type signature. The syntax of the language PLTL($\Sigma$) of extended propositional linear temporal logic over $\Sigma$ has the following Backus Naur Form (BNF) definition:

\[
\phi ::= \bot \mid T \mid s = c \mid s \neq c \mid (\neg \phi) \mid (\phi_1 \land \phi_2) \mid (\phi_1 \lor \phi_2) \mid (\phi_1 \rightarrow \phi_2) \mid \diamond \phi \mid \lozenge \phi \mid \\
\square \phi \mid (\phi_1 U \phi_2) \mid (\phi_1 W \phi_2) \mid (\phi_1 R \phi_2)
\]

where $s \in S$ and $c \in \Sigma_s$.

The atoms of this language consist of equations and in-equations defined over $\Sigma$.

Notice that every type symbol $s \in S$ is also used as the unique variable symbol of that type in PLTL($\Sigma$). This is because these variables are synonyms for the unique ports on the SUT where reading and writing occur. Beyond these single variables PLTL($\Sigma$) has no other first-order features.

The variable $in \in S$ denotes the single input variable for the distinguished input type $in$. Every other type $s \in S$ denotes an output variable for reading values of type $s$. The connectives $\bot, T, \neg, \land, \lor$ and $\rightarrow$ are the usual Boolean connectives with their conventional meanings. The operators $\circlearrowleft, \bigcirc, \square, U, W$ and $R$ are the temporal operators and these were described in Section 5.3.

Semantics of PLTL($\Sigma$)

Let $\Sigma$ be an $s$-sorted finite data type signature and let $A$ be a concrete $\Sigma$ data type. Let $\alpha : N \rightarrow A_i$ be an input sequence. Let $K$ be a deterministic Kripke structure over $A$. Let $i$ be any $i \in N$ then we define $K, \alpha(i) \models \phi$ for $\phi \in PLTL(\Sigma)$ inductively over the structure of $\phi$ as follows:
Finite Data Type Modeling in LBTest

To explain the finite data type modeling supported in LBTest we use the same example of a cruise controller (cc) described in Section 3.2 with the added flavour of data types here. In this case the set of sorts, \( S = \{ \text{in}, \text{mode}, \text{speed}, \text{button} \} \) which gives the four data types of the cc. The data type in is reserved for inputs and as described above is the only data type for this purpose. The remaining three data types mode, speed and button are for outputs. The number of output data types can vary depending upon the SUT being tested. We can keep the input events and their meanings as before to get the signature \( \Sigma_{in} = \{ \text{brake}, \text{dec}, \text{gas}, \text{acc}, \text{button} \} \). To use in LBTest these symbolic inputs are to be encoded in unicode characters which are used by the learning algorithm in LBTest tool. The user however does not need to worry about these encodings later and can still input requirements involving the input data type with the symbolic values in \( \Sigma_{in} \). One possible encoding for the current \( \Sigma_{in} \) is given in Table 6.1. The output data types which in this case are mode, speed and button are shown in the first column of Table 6.2. The second and third columns of this table show the start index (inclusive) and end index (exclusive) in the bit-vector for these output data types. The start indices are 0, 2 and 4 and
### Table 6.2: Output Data Type for cc

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Start Index</th>
<th>End Index</th>
<th>Symbolic Value</th>
<th>Binary Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>0</td>
<td>2</td>
<td>manual</td>
<td>[f, f]</td>
</tr>
<tr>
<td>mode</td>
<td>0</td>
<td>2</td>
<td>cruise</td>
<td>[f, t]</td>
</tr>
<tr>
<td>mode</td>
<td>0</td>
<td>2</td>
<td>disengaged</td>
<td>[t, f]</td>
</tr>
<tr>
<td>speed</td>
<td>2</td>
<td>4</td>
<td>slow</td>
<td>[f, f]</td>
</tr>
<tr>
<td>speed</td>
<td>2</td>
<td>4</td>
<td>cruising</td>
<td>[f, t]</td>
</tr>
<tr>
<td>speed</td>
<td>2</td>
<td>4</td>
<td>fast</td>
<td>[t, f]</td>
</tr>
<tr>
<td>button</td>
<td>4</td>
<td>5</td>
<td>on</td>
<td>[t]</td>
</tr>
<tr>
<td>button</td>
<td>4</td>
<td>5</td>
<td>off</td>
<td>[f]</td>
</tr>
</tbody>
</table>

The end indices are 2, 4 and 5 respectively for `mode`, `speed` and `button` data types. The fourth column of the table lists the symbolic values available for all the data types of column one. For example the data type `mode` has symbolic values `manual`, `cruise` and `disengaged`, the data type `speed` has symbolic values `slow`, `cruising` and `fast` and the data type `button` has symbolic values `on` and `off` respectively. The last column of the table gives the Boolean encodings for all the symbolic values of each data type.

As described in Section 3.2 and as shown in Table 6.2 the output data types represent discretized values. However in a practice data type like the `speed` data type of the cc is represented by an infinite data type such as a floating point type. Such data types must be discretized by means of a discretization formula. The resulting discretization must partition the infinite set of values into a finite set of equivalence classes with precise definition of membership for each of the equivalence class in the partition relation. These discretization formulas must be implemented inside the SUT wrapper (server side) by the tester to ensure correct communication as pointed out in Section 6.3. This also gives LBTest a means to do partition testing. Possible discretization formulas to partition the speed data type into three equivalence classes can be $0.0 \leq \text{speed} < 60.0$ for `slow` speed, $60.0 \leq \text{speed} < 120$ for `cruising` speed and $\text{speed} > 120.0$ for `fast` speed.

Now with the definitions for input and output data types of the cc available to us in Table 6.1 and 6.2 respectively we can describe the requirements for cc in in PLTL(Σ) similar to those described in Section 5.5 for simple PLTL. We consider two informal requirements for the cc and then give their formal description in PLTL(Σ):

1. *When brake is pressed in the cruise mode the vehicle is disengaged.*
   a) $G(\text{mode} = \text{cruise} \& \text{in} = \text{brake} \rightarrow X(\text{mode} = \text{disengaged}))$

2. *When the vehicle goes uphill in the cruise mode with cruising speed its cruising speed is maintained.*
6.3. PROPOSITIONAL LINEAR TEMPORAL LOGIC FOR FINITE DATA TYPES

a) \( G(\text{mode = cruise} \land \text{speed = cruising} \land \text{in = dec} \rightarrow X(\text{speed = cruising})) \)

From the above examples the reader can clearly see that there is only one variable for each data type and there is no support for the quantification over data in the syntax of PLTL(\(\Sigma\)) which makes it very simple to specify requirements in it. At the same time it provides enough syntactic sugar to hide Boolean encodings of symbolic data types.
Chapter 7

Coverage Models in Software Testing

Software Testing is a very important aspect of software development phase, no matter which development methodology is pursued by the system developers. This phase is an attempt to ensure that the delivered software is free of errors and is of a certain acceptable quality. How to quantify the testing activity after testing has been concluded is a challenging question across different communities of testers. For this purpose the concept of coverage is introduced in software testing.

7.1 What is coverage in SW Testing?

The purpose of software testing is to execute a large number of test cases on the SUT to maximize the chances of finding errors. The size of a test suite alone however does not guarantee that an SUT has been tested extensively, for example because several test cases in a test suite may be redundant. To measure or quantify the work achieved by a testing procedure is called coverage. In other words, coverage is the extent to which a specific test activity has achieved its purpose. There is no dearth of software testing methods in the literature and consequently there is no dearth of coverage models for these different testing techniques. From within the broad spectrum of testing techniques, white-box testing and black-box testing are the two major categories of testing methods. The former uses white-box coverage models and the latter uses black-box coverage models to quantify the testing effort.

Several white-box coverage models of software testing have been developed, applied and their results studied by researchers and developers. These include coverage models defined on the basis of control flow, data flow and logic (see [AO08]). However, for black-box testing surprisingly few models of coverage exist. In the following sections we will review structural coverage techniques widely used in white-box testing, then some for black-box testing and finally how we performed an initial study of coverage for our tool LBTest.
7.2 Coverage in White-box Testing

Quantification of software testing in terms of coverage has been well studied in white-box testing. This is because white-box testing is inherently concerned about finding an error at a specific location in the software. Therefore white-box testing is about writing test scenarios from source code and not from requirements. Therefore the goal of this activity is to identify and exercise locations. If there are no loops or recursive calls then the locations will be finite and we can compute a good estimate of how many of them have been exercised. But the problem grows in complexity with the existence of loops in the code because it is difficult to estimate how many times a loop is going to be executed just by looking at the code. The problem is even more compounded when there are branches along with loops in the code because in this case the unfolding of a loop along several different branches has to be considered which makes it even more difficult to compute coverage of locations along all these paths. Nevertheless several white-box coverage techniques following the paradigm of graph coverage (see [AO08]) are available in the literature which include node coverage (NC), edge coverage (EC) and edge-pair coverage (EPC) etc.

In NC, EC and EPC a graph model of the SUT is considered. A test suite achieves NC if its test cases visit each node in the graph. Similarly, a test suite achieves EC or EPC if each reachable path of length $\leq 1$ or length $\leq 2$ in the graph is exercised by its test cases respectively. However, these are not particularly useful when there are loops along some paths in the graph. A more useful coverage approach to deal with loops in the code is called prime paths coverage (PPC) see ([AO08]). A prime path is a path in the graph that can not be extended without losing simplicity. A simple path is a path with no duplicate nodes except (possibly) the first and the last. A test suite will achieve PPC if its test cases exercises each prime path in the graph. There are also some coverage criteria based on logic coverage measures and these include predicate coverage (PC), clause coverage (CC) and MC/DC (see [AO08]). These can be seen as refinements of the graph coverage models.

7.3 Coverage in Requirements-based Black-box Testing

The literature about coverage metrics for black-box testing of software systems is quite scarce. A recent paper about black-box coverage metrics is [WRH06]. In this paper the authors propose some coverage metrics for requirements-based testing. The different criteria used by the authors in [WRH06] are:

- **Requirements coverage**
- **Antecedent coverage**
- **Unique First Cause (UFC) Coverage**
7.4. COVERAGE IN LBTEST

In the first case the requirements formula is negated and a test case is generated by model checking this negated formula against the executable model of the SUT in the model checker’s language.

The second case typically involves requirements formulas of the form containing an implication. The test cases generated from such formulas will be vacuously true if the antecedent is false. Therefore to avoid this scenario a stronger formula is built to generate test cases for antecedent coverage. E.g a formula of the form \( G(A \rightarrow B) \) will be converted to \( G(A \rightarrow B) \land F(A) \) so that test cases without the vacuity condition being true are generated.

The third type of coverage is an extension of modified condition/decision coverage MC/DC criteria described in [CM94, KJDSJJK01] which is a logic coverage metric designed to show the effect of independent Boolean conditions on the Boolean decision (expression) in which they occur. In [WRH06], the authors have adapted MC/DC for LTL properties by modifying its constraints from state-based to path based to include the LTL temporal operators. They call this adaptation the Unique First Cause (UFC) coverage criteria. The tests in a test suite will be said to have fulfilled UFC if for each formula \( f \): i) every basic condition in \( f \) has taken all possible outcomes at least once ii) each basic condition in \( f \) has been shown to affect the outcome of \( f \) independently.

The idea in [WRH06] is to generate test cases from the requirements formulas meeting the above three criteria by model checking an executable model of the SUT. Then these tests are executed on the executable model and different types of model coverage achieved are observed like state coverage, transition coverage and MC/DC coverage.

A probabilistic model of coverage of procedural programs of the form of Hoare triples \( \{p\}S\{q\} \) was introduced in [Mei06]. The purpose of this probabilistic model was to estimate the probability of the correctness of the SUT after the execution of \( n \) test cases on it. However, to extend this approach for automata models and LTL is an open question for future research in this area.

7.4 Coverage in LBTest

Since LBTest is a tool for black-box requirements testing of an SUT then it would have been appropriate to do a coverage analysis of the test cases generated by LBTest on the basis of a well-known black-box coverage metric. However in the absence of such metrics it was decided to follow the approach of [WRH06] to do a simple coverage study for LBTest. It turned out that to use the approach of [WRH06] for coverage analysis of LBTest was not straight forward because the authors had generated a test suite for each coverage criteria before starting testing, while LBTest generates its test cases during testing when it is learning the SUT via random, model checker and active learning queries. What could be done was to perform a graph coverage analysis of LBTest test cases as done in [WRH06] and compare the results obtained with the graph coverage results obtained by [WRH06].
CHAPTER 7. COVERAGE MODELS IN SOFTWARE TESTING

<table>
<thead>
<tr>
<th>Coverage Criteria</th>
<th>% Coverage</th>
<th>No. of TC\textsuperscript{maxCov}</th>
<th>Total TCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>100</td>
<td>149</td>
<td>2437</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>17</td>
<td>2650</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>27</td>
<td>2424</td>
</tr>
<tr>
<td>Transition</td>
<td>100</td>
<td>797</td>
<td>4570</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>634</td>
<td>2740</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>112</td>
<td>2866</td>
</tr>
<tr>
<td>Edge Pair</td>
<td>100</td>
<td>813</td>
<td>2919</td>
</tr>
<tr>
<td></td>
<td>97.5</td>
<td>3370</td>
<td>3370</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>2096</td>
<td>2400</td>
</tr>
<tr>
<td>Prime Paths</td>
<td>44.8</td>
<td>2559</td>
<td>2559</td>
</tr>
<tr>
<td></td>
<td>44.8</td>
<td>2251</td>
<td>2251</td>
</tr>
<tr>
<td></td>
<td>51.7</td>
<td>2997</td>
<td>2997</td>
</tr>
</tbody>
</table>

Table 7.1: Coverage Metrics for LBTest

For this purpose an SUT with source code available was chosen. This replicated the state transition diagram of a simple cruise controller (cc) shown in Figure 3.1. The code was instrumented to record the number of state coverage, transition coverage, edge-pair coverage and prime path coverage. Table 7.1 gives the statistics for each of these graph coverage criteria achieved by LBTest cases. The first column shows the coverage criteria used. Each experiment was repeated three times to cater for the varying influence of random test cases. The results of each of these experiments were recorded in three different rows against each coverage criteria. The second column shows the percentage coverage achieved. The third column depicts the minimum number of test cases (TCs) required to achieve the maximum coverage value for the given criteria and the last column shows the total number of test cases (TCs) to completely learn the SUT.

Analysis of LBTest Results

The results obtained from this coverage study provided some interesting insights about redundancy in existing white-box testing coverage models. We describe the coverage results for each criteria used in the following sections:

State Coverage

In the actual cc state diagram there were 8 states and a key feature of the learning algorithm is that it learns the minimal model of the SUT which is behaviourally equivalent to it. LBTest achieved 100% state coverage in this case in all three experiments. This is because the random queries, model checker queries and active learning queries cover all the states of the SUT.
7.5. PROSPECTS OF REQUIREMENTS COVERAGE

Transition Coverage

This particular SUT for the (cc) has 8 states and the input alphabet set for it consisted of five symbols. Therefore the total number of transitions for the SUT was 40. The third column of Table 7.1 shows the transition coverage achieved by LBTest and it always achieves 100% transition coverage well before completely learning the SUT.

Edge-Pair Coverage

We wrote code to find out all the edge-pair paths in our example SUT to compute edge-pair coverage for LBTest. The number for these paths was 81. But as evident from Table 7.1 LBTest did not always achieve 100% edge-pair coverage despite completely learning the SUT. This result reflects that achieving 100% edge-pair coverage may contain redundant test cases.

Prime Path Coverage

As in the previous case the code for the (cc) SUT was instrumented to compute all the prime paths. The total number of prime paths found was 87. Test cases generated by LBTest at best were able to cover roughly half of these prime paths. The reason seems to be the same as in previous sections that it is not necessary to visit all the prime paths to learn a behaviourally equivalent model of the SUT. This seems to imply that a test suite with 100% prime path coverage will have significant redundancy.

7.5 Prospects of Requirements Coverage

As stated in the preceding sections the problem of coverage for black-box requirements testing following a black-box model of coverage is not well studied in the literature. To estimate the coverage of a black-box SUT is a challenging question because it is difficult to devise an accurate mathematical coverage model of something hidden. On the other hand practical but naive methods of estimating this type of coverage have been un-successful.

However, an interesting prospect in the case of LBTest is that it infers a sequence of hypothesis automata models of the SUT. These inferred models are then tested against a PLTL formula. It might be an interesting idea to reduce the requirements formulas specified in PLTL into some kind of similar automata as the inferred models and then take the intersection of these two automata to estimate the coverage achieved corresponding to the PLTL property being tested.

Another line of action might be to extend the probabilistic model of coverage introduced for procedural programs in [Mei06] to automata models and PLTL properties.
Chapter 8

Software Testing Tools

With the ever decreasing costs and the corresponding increase in quality and performance of computer hardware, the software that runs on it has also grown in complexity substantially. This is mainly in order to fully exploit the features of the underlying hardware. The increase in complexity of the software has caused the existing manual and other related testing approaches for software to achieve less and less coverage. To deliver defect free software these days is an ever increasing challenge for researchers and developers alike. For this goal both researchers and developers have been working on how to automate the testing process to achieve the goal of delivering defect free software. This requires automatically generating a test suite, evaluating tests by giving verdicts automatically and finally reporting test results. With conventional methods of manual or semi automated testing this can be time consuming, error prone and some times even impossible to achieve due to the enormity and complexity of the underlying software system. In this chapter we review some software tools which have emerged from a closely related field of testing model-based testing (MBT). We compare and contrast the features of these tools with our own tool LBTest on the basis of goal, method and outcome achieved by each.

8.1 Testing Tool Survey

The collection of software testing tools is very large and it is unnecessary to survey all of these to understand the results of this thesis. Most relevant are the tools emerging from the MBT community because LBTest also infers its own models to do testing. Table 8.1 lists some MBT tools along with the type of SUTs they test or MBT criteria they follow. We will restrict our attention to tools with similar goals. We will review and compare a non-exhaustive list of such tools with LBTest in terms of methods used and results obtained.

In the following sections we review some closely related MBT tools and describe their approach to automated test case generation. All these tools perform functional
requirements testing which makes it meaningful to compare them with LBTest. However, these tools specify requirements through a reference model following the paradigm of model-based checking and conformance testing. LBTest on the other hand takes PLTL requirements to test an SUT. The use of a reference model is useful in the context that it captures the functionality of the whole SUT, which makes it a global model of all requirements. This approach however is not useful when the intention of the tester is to test a particular use case or requirement which requires a local treatment of properties rather than handling a global model of properties. In LBTest, the SUT is tested against PLTL requirements and each of these represents some local aspect of the SUT which makes LBTest more efficient for use case testing.

LTG/B & LTG/UML

LTG/B and LTG/UML are model-based testing tools from Leirios Smart Testing (see [JL06, BBC+06]). Both of these are meant for functional black-box testing of an SUT. From the specifications of the SUT either a B model [BPR] or UML model [UML] is generated which becomes the basis for test generation. The generated test cases are then translated into executable code tests. These are executed on the SUT and verdicts are assigned on the basis of the expected output provided with each test case. The tester can use different model coverage criteria (like multiple conditions coverage or boundary values coverage) to limit the generated test cases. This also helps to control the test case explosion problem in the absence of these criteria.

Spec Explorer

Spec Explorer is a model-based testing tool from Microsoft (see [VCG+08]) which also works on the principle of conformance testing. Spec Explorer encodes the specification of the SUT in a machine executable format (which becomes the model of the SUT replicating an abstract state machine). The Spec Explorer tool then explores the possible runs of this model to systematically generate test cases. The

<table>
<thead>
<tr>
<th>SUT Type / MBT Criteria</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real Time SUTs</td>
<td>RT-Tester[DMP00], Uppaal-Tron[LMNS05, HLM+08]</td>
</tr>
<tr>
<td>Concurrent SUTs</td>
<td>Agatha [BFG+03], Agedis [HN05]</td>
</tr>
<tr>
<td>Probabilistic Criteria</td>
<td>MaTeLo [DZ03], JUMBL [Pro03]</td>
</tr>
<tr>
<td>Structural Coverage Criteria</td>
<td>TPT [BK08], BTC Embedded Tester[BTC], Reactis Tester[Rea], MTest [MTe]</td>
</tr>
<tr>
<td>Transition System SUTs</td>
<td>TorX [TBR03], STG [CJRZ02]</td>
</tr>
</tbody>
</table>

Table 8.1: Testing Tools from the MBT Community
behaviour of the model program is compared against the actual SUT behaviour to find violations of conformance relation. A conformance error may be: (i) an implementation error (which is an error in the SUT), (ii) a modeling error (which is an error in the generated model from the specification), (iii) a specification error (which is a wrong description of the intended behaviour) or simply (iv) a design error (which constitutes a logical inconsistency in the SUT’s intended behaviour).

The verdict given by Spec Explorer on test cases can be of the types: (i) Undecided (ii) Succeeded (iii) Failed and (iv) TimedOut. A test outcome remains undecided if it can not be computed to be Succeeded, Failed or TimedOut. A test is succeeded when the observed and expected outcomes match otherwise it is failed. A test is TimedOut if no observable outcome is seen by the tester during the time allocated by the tester.

**STG**

STG or Symbolic Test Generator is a test generation tool for black-box SUTs (see [CJRZ02]). This tool also works on the principle of conformance testing which is a methodology to validate the behaviour of reactive systems. One key feature of this tool is the use of IOSTS which stands for Input/Output Symbolic Transition Systems. The IOSTS constitute reactive system models which work by symbolic processing of variables and inter-process value passing. The IOSTS are used to describe specifications, test purposes and test cases themselves but assume nothing about the black-box SUT except that it is interface compatible with the specification. The test cases are computed from all the possible behaviours in the intersection of a specification and a test process. The test cases are then translated into executable code to be run on the black-box implementation. Three different verdicts are given depending upon the observed behaviour and the conformance relation and these can be a pass, Inconclusive or a fail. A pass means that the SUT behaviour was in conformance with the specification and the goal of test case was achieved, an inconclusive verdict means that the SUT behaved correctly but due to the lack of control on the SUT the desired goal was not achieved and the fail verdict simply shows a non-conforming behaviour.

**UPPAAL-TRON**

Uppaal-Tron is a testing tool for black-box conformance testing of real time embedded software (see [LMNS05, HLM+08]). The emphasis of this tool is on testing timed and functional properties. The tool does this by checking whether the timed runs of the SUT match the specified system model without encountering illegal timed behaviour. The specification is described in terms of a timed automata network which is partitioned into a system model and its environment. A distinct feature is to handle system implementations with soft time deadlines which is necessary because the models can have non-determinism in them. The conformance of real-time systems is generally undecidable but in the case of Uppaal-Tron it has
been made decidable after making some digitization assumptions. Moreover, it is shown to be sound and complete. Soundness of conformance means that if conformance does not hold then there is something wrong with the SUT (indicates presence of error). The completeness of conformance means that if there is a bug in the SUT then an error trace will be revealed and the conformance will not hold. The tool has been applied to industrial case studies (such as on an advanced electronic thermostat from Danfoss) with effectiveness (see [LMNS05]).

JUMBL

JUMBL is a tool for model-based statistical testing (see [Pro03]). The word JUMBL stands for J Usage Model Builder Library. It also consists of command line tools to work with usage models. It develops a usage model of the SUT based on statistical techniques. A usage model is a description stating how a system is used rather than what it will do. Alternatively it is a mechanism to predict what the user will try to do and not how the system behaves in response to user actions. This model can then be used to generate test cases depending upon expected usage of the SUT.

It supports construction and analysis of usage models, test case generation and execution along with verdict construction by analyzing test results.

LBTest

LBTest is a tool for learning-based testing of reactive systems. A detailed description of the tool can be seen in Part II, Paper 3. Here we provide the salient features of the tool for comparison with other tools cited above. In LBTest, black-box functional requirements are tested by implementing an incremental learning algorithm which is integrated with a PLTL model checking algorithm. The learning algorithm generates a sequence of hypothesis automata $H_0, H_1, H_2, \ldots$. Each of these is model checked against a PLTL requirements formula to find any counterexample in the behaviour of the SUT. In the case where a counterexample is found it is executed on the last hypothesis and the SUT to get the predicted and observed outputs respectively. Both the predicted and observed outputs are compared by the oracle to give a verdict. The verdict can be a pass, a warning or a fail. A pass verdict is given when the expected and observed outputs do not match and the learning algorithm continues learning a new hypothesis. A fail verdict is given when both the observed and predicted outputs match showing a discrepancy in the learned model of the SUT. A warning verdict is given when there is a loop in the counterexample and the observed and expected outputs do not match. This deals with the case where the requirements formula is a liveness property with no finite counterexample. In the case where no counterexample is found by the model checker LBTest continues learning the next hypothesis from a randomly generated input string.
<table>
<thead>
<tr>
<th>Tool</th>
<th>Pre-existing Model Required</th>
<th>Model used / Inferred</th>
<th>Model Update during Testing</th>
<th>Verdicts Supported</th>
<th>Coverage Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTG/B &amp; LTG/UML</td>
<td>Yes</td>
<td>B &amp; UML</td>
<td>No</td>
<td>None</td>
<td>transition-based, decision-based, data-oriented</td>
</tr>
<tr>
<td>Spec Explorer</td>
<td>Yes</td>
<td>ASM</td>
<td>No</td>
<td>Undecided, Succeeded, Fail, TimedOut</td>
<td>node-based, transition-based</td>
</tr>
<tr>
<td>STG</td>
<td>Yes</td>
<td>IOSTS</td>
<td>No</td>
<td>Pass, Inconclusive, Fail</td>
<td>None</td>
</tr>
<tr>
<td>Uppaal-Tron</td>
<td>Yes</td>
<td>Timed Automata</td>
<td>No</td>
<td>Pass, Fail</td>
<td>None</td>
</tr>
<tr>
<td>Jumbl</td>
<td>Yes</td>
<td>Usage Model</td>
<td>No</td>
<td>Pass, Fail</td>
<td>Probabilistic</td>
</tr>
<tr>
<td>LBTest</td>
<td>No</td>
<td>Deterministic Kripke structures</td>
<td>Yes</td>
<td>Pass, Warning, Fail</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 8.2: Testing Tool Comparison
8.2 Tool Comparison

We gave a review of several software testing tools in the preceding sections including our own tool LBTest. LBTest is the first tool of its kind which constructs its own models during testing unlike the other tools reviewed here, which require a model before starting testing. Since test cases are generated from the model as in other tools, these tools to some extent are comparable in goal, method and result.

A comparison of LBTest and the tools reviewed above is summarized in Table 8.2. To have a model available before starting testing limits a tool in two ways: 1) the larger the model more time / resources it will take before testing is started; 2) in case a change is required either in the SUT or the implementation the whole modeling phase has to be re-done to manage this change which makes it suitable for agile development. In the case of LBTest these hinderances are absent because we do not require a model to start testing. Also models in LBTest are built incrementally and updated automatically with each iteration freeing the testers from managing changes in models / SUTs.
Chapter 9

Conclusions and Future Work

In Part I so far we have reviewed essential concepts to understand the papers appearing in Part II of this thesis. Now in this chapter we first give a brief summary of Part I followed by some conclusions about the work presented and finally we give some future directions of research.

9.1 Summary

In chapters 2-8 we have provided an overview of this thesis and 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 in Chapter 2. 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. However, as they have a common goal and complementary nature there is a need to develop an “intermediate” approach. This approach should exploit the complementary nature of both testing and verification. From this point of view learning-based testing combines verification techniques such as model checking 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 4 and an overview of model checking and temporal logics was given in Chapter 5.

Requirements languages form the back-bone of any requirements-based testing. Therefore, an overview of requirements languages in the context of formal methods was given in Chapter 6. This chapter also gave an introduction to the PLTL($\Sigma$) language which is the requirements language of the LBTest tool.

A key element for any software testing activity is to quantify the testing effort no matter which testing approach is used. In this context, coverage models corresponding to different types of testing were discussed in Chapter 7. This chapter
also describes a simple coverage study carried out with LBTest.

The novel approach introduced in this thesis combines the features of both testing and verification to test systems. This was achieved practically by integrating a verification tool (NuSMV model checker) with an incremental regular inference algorithm (IKL) for multi-bit output to generate test cases to realize the LBTest tool. Since LBTest does requirements testing of functional properties of inferred models, therefore, a review of similar tools following the paradigm of model-based testing was provided in Chapter 8.

9.2 Contributions of the thesis

The contribution of this thesis to the field can be described at four different levels:

1. At an abstract level we designed, verified and developed algorithms for incremental learning of DFA and Kripke structures. These algorithms made use of the concepts of distinguishing sequences, lazy partition refinement, sub-direct product construction and automata minimization to be practically used inside a real tool for testing.

2. At an implementation level the whole learning-based testing architecture was instrumented with the capabilities of random querying, constructing an oracle that could automatically give verdicts on generated test cases and enabling it to give judgements on both safety and liveness properties.

3. At the actual tool level features were built to take use case scenarios as input with the help of an appropriate requirements language, information management of the generated test cases (both on screen and on disk) and visualization of generated automata during the testing activity along with displaying verdicts visually.

4. At the user level the tool was evaluated first on the basis of running personal case studies on it. Later, it was handed over to other users to evaluate its feasibility as a transferable technology.

On the basis of the above points which summarize the historical development of our research papers, we can conclude that learning-based testing is an effective methodology and LBTest is a promising tool for both pedagogical and industrial case studies with a high potential for becoming an easily transferable technology in future.

9.3 Author’s personal contribution

The work on this thesis led to the following papers:
9.3. AUTHOR’S PERSONAL CONTRIBUTION

Paper 1

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.

Paper 2
K. Meinke and M. A. Sindhu, *An Efficient Model Inference Algorithm for Learning-based Testing of Reactive Systems*, a preliminary version of this paper was accepted (after a peer review) for a poster presentation in 3rd Asian Conference on Machine Learning (ACML 2011). An extended version has been submitted to the journal, “Software Tools for Technology Transfer.”

The algorithms presented in this paper were jointly developed during discussions with my supervisor. I also worked on some parts of proof of correctness. I did all the implementation and experimental evaluation of this framework.

Paper 3

The tool presented in this paper was developed by me. Sections 3-6 of the paper were also written by me.

Paper 4

I interacted with all the authors of this paper. I set up the case studies to be tested with LBTest, performed all the experiments, obtained results and analyzed them. I also wrote some parts of Section 4 and all of section 5 of this paper.

Other Papers

9.4 Future Work

In the short term we envisage to do several optimizations to the current *LBT* est tool. 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.

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

• Current practice of software testing in the industry does not commonly follow the paradigm of formal requirements let alone requirements written in temporal logic. We therefore, plan to extend the requirements language of LBTest to a more user friendly graphical user requirements language. Extracting temporal logic formulas from graphical state charts (UML or otherwise) have been studied in the literature (see [GM05]) which can become the basis of this direction of research.

• LBTest is a generic methodology requiring a learning algorithm to infer models which can be model checked by a model checker to generate test cases. This methodology can be used for different types of systems depending upon the availability of relevant learning and model checking algorithms. For example SUTs with hybrid / realtime behaviour can also be tested with the same framework depending upon the availability of learning and model checking algorithms for hybrid / real-time automata. In which case it will just be a matter of integrating them in the current LBTest tool. The same is true for testing of SUTs involving infinite data types (see e.g. [MN11]).

• Even for Boolean SUTs it will be interesting to see how different learning algorithms enhance or impede testing. This is especially relevant in the context when some learning algorithms need fewer membership queries, others need more and some don’t need them at all, as described in Chapter 4. The effect of these on the testing activity still needs to be studied. Therefore, integration of more learning algorithms into LBTest is planned for future releases of the tool.

• Coverage for black-box requirements-based testing following a black-box coverage model is not well studied in the literature. This can be another interesting direction of research i.e. to enhance LBTest with a suitably defined
coverage model for black-box requirements testing. For example the probabilistic coverage model for procedural systems introduced in [Mei06] could be studied to extend for automata models and LTL properties.
Bibliography


[MTe] MTtest (http://mttest-classic.com/).


[Rea] Reactis (http://www.reactive-systems.com/).


Part II

Included Papers
Appendix A

Paper 1 (Incremental Learning-based Testing for Reactive Systems)
Incremental Learning-Based Testing for Reactive Systems

Karl Meinke, Muddassar. 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 \textit{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 \textit{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 $\text{Req}$ 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 \textit{heuristic iterative methods} to automatically generate a sequence of test cases. The heuristic approach is based on \textit{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 \textit{incremental learning algorithm} (see Section 1.2). This step involves \textit{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 $\text{Req}$ are satisfiability checked against the learned model $M_n$ derived in Step 1 (aka. \textit{model checking}). This process searches for a \textit{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 $\text{Req}$) then $i_{n+1}$ was a \textit{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 \textit{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 $M_0, M_1, M_2, \ldots$, 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 $\text{Req}$ as soon as possible.

An automata learning algorithm $L$ is said to be incremental if it can produce a sequence of hypothesis automata $A_0, A_1, \ldots$ which are approximations to an unknown automata $A$, based on a sequence of information (queries and results) about $A$. The sequence $A_0, A_1, \ldots$ 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 $\text{Req}$.

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 $\text{Req}$. 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 $\text{Req}$, 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 practice, 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]) which learn Mealy automata. To our knowledge, no incremental 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 \( \Sigma \) be any set of symbols then \( \Sigma^* \) denotes the set of all finite strings over \( \Sigma \) including the empty string \( \varepsilon \). The length of a string \( \alpha \in \Sigma^* \) is denoted by \( |\alpha| \) and \( |\varepsilon| = 0 \). For strings \( \alpha, \beta \in \Sigma^* \); \( \alpha \cdot \beta \) denotes their concatenation.

For \( \alpha, \beta, \gamma \in \Sigma^* \), if \( \alpha = \beta \gamma \) then \( \beta \) is termed a prefix of \( \alpha \) and \( \gamma \) is termed a suffix of \( \alpha \). We let \( \text{Pref}(\alpha) \) denote the prefix closure of \( \alpha \), i.e. the set of all prefixes of \( \alpha \). We can also apply prefix closure pointwise to any set of strings. The set difference operation between two sets \( U, V \), denoted by \( U \setminus V \), 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 \( U \oplus V = (U \setminus V) \cup (V \setminus U) \).

A deterministic finite automaton (DFA) is a five-tuple \( A = \langle \Sigma, Q, F, q_0, \delta \rangle \) where: \( \Sigma \) 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 \( \delta : Q \times \Sigma \rightarrow Q \) with the usual meaning, and can be inductively extended to a mapping \( \delta^\ast : Q \times \Sigma^* \rightarrow Q \) where \( \delta^\ast(q, \varepsilon) = q \) and \( \delta^\ast(q, \sigma_1, \ldots, \sigma_{n+1}) = \delta(\delta^\ast(q, \sigma_1, \ldots, \sigma_n), \sigma_{n+1}) \). 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 \( d_0 \), \( f : \Sigma^* \cup \{d_0\} \times \Sigma \rightarrow \Sigma^* \cup \{d_0\} \), by \( f(d_0, \sigma) = d_0 \) and \( f(\alpha, \sigma) = \alpha \cdot \sigma \) 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 \( \delta^\ast(q_0, \alpha) \in F \). 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

\[
A = (Q_A, \Sigma, \delta_A : Q_A \times \Sigma \rightarrow Q_A, q_A^0, \lambda_A : Q_A \rightarrow \mathbb{B}^k)
\]
3.2. Definition. Furthermore, if $t$ and $A_{\delta}$, for such structures we have $t$, and only if, for every finite input sequence $\sigma$, $\sigma_{i} \in \Sigma^{*}$ 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 $\Sigma$, $A = ( Q_{A}, \Sigma, \delta_{A} : Q_{A} \times \Sigma \rightarrow Q_{A}, q_{A}^{0}, \lambda_{A} : Q_{A} \rightarrow 2^{k})$. 

where $Q_{A}$ is a state set, $\delta_{A}$ is the state transition function, $q_{A}^{0}$ is the initial state and $\lambda_{A}$ is the output function. As before we let $\delta_{A} : Q_{A} \times \Sigma^{*} \rightarrow Q_{A}$ 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 $\lambda_{A}^{*} : \Sigma^{*} \rightarrow 2^{k}$ 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 $A' = ( Q_{A}, \Sigma, \delta_{A} : Q_{A} \times \Sigma \rightarrow Q_{A}, q_{A}^{0}, F_{A'} )$, where $F_{A'} \subseteq Q_{A}$ and $\lambda_{A}(q) = true$ 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 $\Sigma$. 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
For each $1 \leq i \leq k$ define the $i$-th projection $A_i$ of $A$ to be the 1-bit Kripke structure where

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

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,

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

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

$$\text{proj}_i : Q_1 \times \ldots \times Q_k \to Q_i, \quad \text{proj}_i(q_1, \ldots, q_k) = q_i, \quad 1 \leq i \leq k$$

Let $\text{Min}(\prod_{i=1}^k A_i)$ 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 $\text{proj}_i$ is a homomorphism.

3.4. Proposition. Let $A_1, \ldots, A_k$ be any minimal 1-bit Kripke structures.

(i) For each $1 \leq i \leq k$, $\text{proj}_i : \text{Min}(\prod_{i=1}^k A_i) \to A_i$ is an epimorphism, and hence $\text{Min}(\prod_{i=1}^k A_i)$ is a subdirect product of the $A_i$.

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

Proof. (i) Immediate since the $A_i$ are minimal. (ii) Follows from $\text{Min}(A) \equiv A$. 
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 \( \Sigma \). 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, \( A_1 = B_1 \land \ldots \land A_k = B_k \) then

\[
A = \operatorname{Min}(\prod_{i=1}^{k} B_i).
\]

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 \( E_j^i \), a set \( V_j \) of distinguishing strings is generated iteratively to split pairs of equivalence classes in \( E_j^i \) until a congruence is achieved. Then a quotient DFA \( M^j \) can be constructed from the partition of \( T_k \) by \( E_j^i \). The congruences are constructed so that \( E_j^i \subseteq E_{j+1}^i \) 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 \( M_1^1, \ldots, M^n_1 \) (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 \( \phi \) expressed in temporal logic. If no counterexamples to \( \phi \) 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 \( M_1^1, \ldots, M^n_1 \).
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 $\lambda_A(s_i) = \top_	op$.

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. $(b_1, \ldots, b_n) = \lambda_A^c(\alpha)$
11. for $c = 1$ to $n$ do
12. if $b_c$ then $E^c_0(\alpha) = \{v_{i_c}\}$ else $E^c_0(\alpha) = \emptyset$
13. }
14. //Refine the initial equivalence relations $E^1_0, \ldots, E^n_0$
15. //into congruences using Algorithm 2
16. //Synthesize an initial Kripke structure $M_0$ approximating $A$
17. //using Algorithm 3.
18. //Process the file of examples.
19. while $S \neq \emptyset$ do {
20. read($S, \alpha$)
21. $k = k+1$, $t = t+1$
22. $P_k = P_{k-1} \cup \text{Pref}(\alpha)$ //prefix closure
23. $P'_k = P_k \cup \{d_0\}$
24. $T_k = T_{k-1} \cup \text{Pref}(\alpha) \cup \{\alpha \cdot b \mid \alpha \in P_k - P_{k-1}, b \in \Sigma\}$ //for prefix closure
25. $T'_k = T_k \cup \{d_0\}$
26. $\forall \alpha \in T_k - T_{k-1}$ { for $c = 1$ to $n$ do $E^c_0(\alpha) = \emptyset$ //initialise new equivalence class $E^0_0(\alpha)$
27. for $j = 0$ to $i_c$ do {
28. // Consider adding distinguishing string $v_j \in V_c$
29. // to each new equivalence class $E^c_j(\alpha)$
30. if $b_c$ then $E^c_j(\alpha) = E^c_j(\alpha) \cup \{v_j\}$
31. }
32. //Refine the current equivalence relations $E^1_1, \ldots, E^n_{i_n}$
33. //into congruences using Algorithm 2
34. if $\alpha$ is consistent with $M_{t-1}$
35. then $M_t = M_{t-1}$
36. else synthesize Kripke structure $M_t$ using Algorithm 3.
37. }
38. end.
Algorithm 2 Lazy Partition Refinement

1. while (\( \exists 1 \leq c \leq n, \exists \alpha, \beta \in P_k^c \) and \( \exists \sigma \in \Sigma \) such that \( E^c_{\alpha}(\sigma) \neq E^c_{\beta}(\sigma) \) but \( E^c_{\alpha}(f(\alpha, \sigma)) \neq E^c_{\beta}(f(\beta, \sigma)) \)) do {
   2. // Equivalence relation \( E^c_{\alpha} \) is not a congruence w.r.t. \( \delta_c \)
   3. // so add a new distinguishing sequence.
   4. Choose \( \gamma \in E^c_{\alpha}(f(\alpha, \sigma)) \oplus E^c_{\beta}(f(\beta, \sigma)) \)
   5. \( v = \sigma \cdot \gamma \)
   6. \( \forall \alpha \in T_k \{
   7. \quad (b_1, \ldots, b_n) = \lambda^\alpha(\alpha \cdot v)
   8. \quad \text{for } c = 1 \text{ to } n \text{ do } \{
   9. \quad \text{if } E^c_{\alpha}(\alpha) = E^c_{\beta}(\beta) \text{ and } E^c_{\alpha}(f(\alpha, \sigma)) \neq E^c_{\beta}(f(\beta, \sigma)) \text{ then } \{
   10. \quad \quad \text{// Lazy refinement of equivalence relation } E^c_{\alpha}\n   11. \quad \quad i_c = i_c + 1, v_{i_c} = v, V_c = V_c \cup \{v_{i_c}\}\n   12. \quad \quad \text{if } b_c \text{ then } E^c_{\alpha}(\alpha) = E^c_{\alpha-1}(\alpha) \cup \{v_{i_c}\} \text{ else } E^c_{\alpha}(\alpha) = E^c_{\alpha-1}(\alpha)\n   13. \quad \}\n   14. \}\n   15. \}\n
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 \( E^c_{\alpha}(\alpha) \), where \( \alpha \in T_k \)
   4. Let \( q^0 = E^c_{\alpha}(\varepsilon) \)
   5. The accepting states are the sets \( E^c_{\alpha}(\alpha) \) where \( \alpha \in T_k \) and \( \varepsilon \in E^c_{\alpha}(\alpha) \)
   6. The transition function \( \delta_c \) of \( M^c \) is defined as follows:
   7. \( \forall \alpha \in P_k^c \{
   8. \quad \text{if } E^c_{\alpha}(\alpha) = \emptyset \text{ then } \forall b \in \Sigma \{ \text{ let } \delta_c(E^c_{\alpha}(\alpha), b) = E^c_{\alpha}(\alpha) \}\n   9. \quad \text{else } \forall b \in \Sigma \{ \delta_c(E^c_{\alpha}(\alpha), b) = E^c_{\alpha}(\alpha \cdot b) \}\n   10. \}\n   11. \forall \beta \in T_k - P_k^c \{
   12. \quad \text{if } \forall \alpha \in P_k^c \{ E^c_{\alpha}(\beta) \neq E^c_{\alpha}(\alpha) \} \text{ and } E^c_{\alpha}(\beta) \neq \emptyset \text{ then }
   13. \quad \forall b \in \Sigma \{ \delta_c(E^c_{\alpha}(\beta), b) = \emptyset \}\n   14. \}\n   15. // Compute \( M_t \) in linear time as a subdirect product of the \( M^c \)
   16. \( M_t = M_m( \prod_{c=1}^n M^c ) \)
Thus line 1 searches for congruence failure in any one of the equivalence relations $E^i_1, \ldots, E^i_n$. 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 $E^j_i$ that is not yet a congruence. On the other hand, any equivalence relation $E^j_i$ that is already a congruence is not refined, even though the result $b_j$ of the new query $\alpha \cdot v$ 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 $M_l$ 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 $A = \langle \Sigma, Q, F, q_0, \delta \rangle$ 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 $\Sigma$. 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 structures 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.)

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 $t_{\text{total}}$ when we have learned the entire SUT (c.f. Section 6). Therefore the equivalence checker allows us to compute the time $t_{\text{total}}$ 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

![Figure 1. A Learning-Based Testing Architecture using the IKL algorithm.](image-url)
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 $\phi$. This formula is constant during a particular testing experiment. The model checker attempts to identify at least one counterexample to $\phi$ in $M_n$ as an input sequence $i$. If $\phi$ is a safety formula then this input sequence will usually be finite $i = i_1, \ldots, i_k$. If $\phi$ is a liveness formula then this input sequence $i$ may be finite or infinite. Recall that infinite counterexamples to liveness formulas can be represented as infinite sequences of the form $x^\omega$. In the case that $i = x^\omega$ then $i$ is truncated to a finite initial segment that would normally include the handle $x$ and at least one execution of the infinite loop $y^\omega$, such as $i = x y^\omega$ or $i = x y^\omega y$. 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 $i$ 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 $i$ 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 $i$, 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 $i = i_1, \ldots, i_k$ is constructed. Figure 1 shows that if $i$ is obtained by model checking then the current model $M_n$ is applied to $i$ to compute a predicted output $p = p_1, \ldots, p_k$ for the SUT that can be used for the oracle step. However, this is not possible if $i$ is random or a partition refinement since then we do not know whether $i$ is a counterexample to $\phi$. Nevertheless, in all three cases, the input sequence $i$ is passed to the SUT and executed to yield an actual or observed output sequence $o = o_1, \ldots, o_k$.

The final stage of this iterative testing architecture is the oracle step. Figure 1 shows that if a predicted output $p$ exists (i.e. the input sequence $i$ came from model checking) then actual output $o$ and the predicted output $p$ are both passed to an oracle component. This component implements the Boolean test $o = p$. If this equality test returns $true$ and the test case $i = i_1, \ldots, i_k$ was originally a finite test case then we can conclude that the test case $i$ is definitely failed, since the behaviour $p$ is by construction a counterexample to the correctness of $\phi$. If the equality test returns $true$ and the test case $i$ 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 $o$ and the incorrect behaviour $p$. The system tester is thus encouraged to consider a potential SUT error.

On the other hand if $o \neq p$, or if no output prediction $p$ exists then it is quite difficult to issue an immediate verdict. It may or may not be the case that the observed output $o$ is a counterexample to the correctness of $\phi$. In some cases the syntactic structure of $\phi$ is simple enough to semantically evaluate the formula
on the fly with its input and output variables bound to $\vec{i}$ and $\vec{o}$ respectively. However, sometimes this is not possible since the semantic evaluation of $\phi$ 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 $\vec{o}$. Model checking $M_{n+1}$ later on will confirm $\vec{o}$ 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 $\Sigma$.

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

$$M_f \equiv A.$$

(ii) If there exists a (finite or infinite) input string $\vec{i}$ over $\Sigma$ which witnesses that an LTL requirement $\phi$ is not valid for $A$, then model checking will eventually find such a string $\vec{i}$ and the LBT architecture will generate a test fail or test warning message after executing $\vec{i}$ 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 $M_f$ with the SUT will succeed and the LBT architecture will terminate.
(ii) Suppose there is at least one (finite or infinite) counterexample string \( \bar{i} \) over \( \Sigma \) to the validity of an LTL requirement \( \phi \) 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 \( \bar{i} \) from the final hypothesis structure \( M_f \), since by part (i), \( M_f \equiv A \) and \( A \) has a counterexample. For such \( \bar{i} \), comparing the corresponding predicted output \( \bar{p} \) from \( M_f \) and the observed output \( \bar{o} \) from \( A \) we must have \( \bar{p} = \bar{o} \) since \( M_f \equiv A \). 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 \( t_{\text{first}} \) at which an error was discovered in an SUT, and to compare this with the total time \( t_{\text{total}} \) required to completely learn the SUT. (So \( t_{\text{first}} \leq t_{\text{total}} \).) 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.

Table 1. Cruise Controller Requirements as LTL formulas.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>$t_{\text{first}}$ (sec)</th>
<th>$t_{\text{total}}$ (sec)</th>
<th>$MCQ_{\text{first}}$</th>
<th>$MCQ_{\text{total}}$</th>
<th>$PQ_{\text{first}}$</th>
<th>$PQ_{\text{total}}$</th>
<th>$RQ_{\text{first}}$</th>
<th>$RQ_{\text{total}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req 1</td>
<td>3.5</td>
<td>21.5</td>
<td>3.2</td>
<td>24.3</td>
<td>7383</td>
<td>30204</td>
<td>8.2</td>
<td>29.3</td>
</tr>
<tr>
<td>Req 2</td>
<td>2.3</td>
<td>5.7</td>
<td>5.5</td>
<td>18.2</td>
<td>8430</td>
<td>27384</td>
<td>10.4</td>
<td>23.1</td>
</tr>
<tr>
<td>Req 3</td>
<td>2.3</td>
<td>16.0</td>
<td>1.7</td>
<td>33.7</td>
<td>6127</td>
<td>34207</td>
<td>6.8</td>
<td>38.8</td>
</tr>
<tr>
<td>Req 4</td>
<td>2.9</td>
<td>6.1</td>
<td>4.7</td>
<td>20.9</td>
<td>7530</td>
<td>24566</td>
<td>10.4</td>
<td>20.9</td>
</tr>
</tbody>
</table>

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 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).](image)

| Req 1 | $G( \text{Stop} \rightarrow ( \oplus 1 | \oplus 2 | \oplus 3 ) )$ |
| Req 2 | $G( \text{Stop} \rightarrow \text{cl} )$ |
| Req 3 | $G( \text{Stop} \& \text{!Stop} \rightarrow \text{X( !cl )} )$ |
| Req 4 | $G( \text{Stop} \& \oplus 1 \& \text{cl} \& \text{in=c1} \& \text{X( @1 )} \rightarrow \text{X( !cl )} )$ |
| Req 5 | $G( \text{Stop} \& \oplus 2 \& \text{cl} \& \text{in=c2} \& \text{X( @2 )} \rightarrow \text{X( !cl )} )$ |
| Req 6 | $G( \text{Stop} \& \oplus 3 \& \text{cl} \& \text{in=c3} \& \text{X( @3 )} \rightarrow \text{X( !cl )} )$ |

**Table 3.** Elevator Requirements as LTL formulas.
Table 4. LBT performance for Elevator Requirements.

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

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

Appendix B

Paper 2 (An Efficient Model Inference Algorithm for Learning-based Testing of Reactive Systems)
An Efficient Model Inference Algorithm for Learning-based Testing of Reactive Systems

Karl Meinke, Muddassar 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. Learning-based testing (LBT) is an emerging methodology to automate iterative black-box requirements testing of software systems. The methodology involves combining model inference with model checking techniques. However, a variety of optimisations on model inference are necessary in order to achieve scalable testing for large systems. In this paper we describe the IKL learning algorithm which is an active incremental learning algorithm for deterministic Kripke structures. We formally prove the correctness of IKL. We discuss the optimisations it incorporates to achieve scalability of testing. We also evaluate a black box heuristic for test termination based on convergence of IKL learning.

1 Introduction

A heuristic approach to automated test case generation (ATCG) from formal requirements specifications known as learning-based testing (LBT) was introduced in [13], [14] and [17]. 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 LBT is to automatically generate a large number of high-quality test cases by combining a model checking algorithm with an optimised model inference algorithm (aka. learning algorithm). For both procedural ([14]) and reactive systems ([15], [17]) it has been shown that LBT can significantly outperform random testing in the speed with which it finds errors in a system under test (SUT). This is because random test suites generally contain a large degree of redundancy, which can be reduced by using learning algorithms and model checkers to execute a more directed search for software errors.

An efficient and practical implementation of learning-based testing for reactive systems has been developed in the LBTest tool [18]. In this paper we describe the IKL (Incremental Kripke Learning) algorithm implemented in LBTest. IKL is an algorithm for active incremental learning of deterministic Kripke structures. The reliability of LBTest for producing correct test results depends crucially on the correctness of this learning algorithm. So we give a formal definition of IKL and prove its correctness. The IKL algorithm involves a number of optimisations necessary to achieve scalability of testing for large software systems. We discuss these optimisations from the perspective of learning and testing.
The problems of coverage, and termination criteria for black-box testing, are complex and different solutions have been proposed. In LBT, convergence of learning can sometimes be used as a criterion to terminate testing. However, heuristics are needed to estimate convergence in the context of black box testing. We will empirically evaluate the reliability of a simple heuristic for IKL.

In the remainder of Section 1, we discuss the general paradigm of LBT, and specific requirements on learning for efficient testing of reactive systems. In Section 2, we review some essential mathematical preliminaries. In Section 3, we present the architecture of the IKL learning algorithm and its main components. These three main components are defined and analysed in detail in Sections 4, 5 and 6. In Section 4, we consider a learning algorithm for families of DFA which supports incremental learning and projection (to be discussed in Section 1.2). In Section 5, we consider integrating a family of DFA into a single Kripke structure using a subdirect product construction. In Section 6, we consider an efficient minimisation algorithm for deterministic Kripke structures based on Hopcroft’s DFA minimisation algorithm [12]. This is needed by the IKL algorithm to produce hypothesis models that can be efficiently model checked. In Section 7, we empirically evaluate a black box heuristic to detect convergence of IKL, that can be used as a test termination criterion. Finally, in Section 8 we draw some conclusions and suggest prospects for further research on learning and testing.

1.1 Learning-Based Testing

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 testing is a heuristic iterative method to automatically generate a sequence of test cases. The heuristic concept is to learn a black-box system using tests as queries.

In general, 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 observations $(i_1, o_1), \ldots, (i_n, o_n)$ can be synthesized into a learned model $M_n$ of $S$ using an incremental learning algorithm (see Section 1.2). This step involves generalization from the observed behaviour, (which represents an incomplete description of $S$) to all possible behaviour. This generalisation step gives the possibility to predict previously unseen errors in $S$ during Step 2.

(Step 2) The system requirements $Req$ are 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
observation \((i_{n+1}, o_{n+1})\) 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. Practical criteria for termination of testing include a bound on the maximum testing time, or a bound on the maximum number of test cases to be executed. However, it also seems possible to derive more theoretically well-founded criteria for termination based on learning theory. One simple approach will be discussed in Section 7. A more sophisticated proposal can be found in [25].

This iterative approach to automated test case generation yields a sequence of increasingly accurate models \(M_0, M_1, M_2, \ldots\) of \(S\). (We usually take \(M_0\) to be a null hypothesis about \(S\).) So, with increasing values of \(n\), it becomes more and more likely that model checking in Step 2 will produce a true negative if one exists.

Notice, if Step 2 does not produce any counterexamples at all then to proceed with the next iteration, we must construct the next test case \(i_{n+1}\) by some other method. Now active learning algorithms can be devised to generate queries that efficiently learn an unknown system in polynomial time. So for LBT there is clearly an advantage to combine model checking with active learning and generate both types of test cases. More generally, it is useful to have access to as wide a variety of query generation techniques as possible. So in practice, model checker and active learning queries are augmented with random queries when necessary. However, these different types of queries need to be combined carefully to achieve efficient and scalable testing.

### 1.2 Learning for Efficient Testing

As has already been suggested in Section 1.1, for LBT to be effective at finding errors, it is important to use the right kind of learning algorithm. As well as active learning, several other principles for efficient testing can be found. To motivate the design of the IKL algorithm we will discuss two of them. For this purpose, we focus specifically on automata learning for testing reactive systems. (LBT has also been successfully applied to testing other types of systems, see e.g. [14]). Learning algorithms for automata are also known as regular inference algorithms in the literature (e.g. [8]).

**Incremental Learning** One efficiency principle is that a good learning algorithm should maximise the opportunity of the model checker in Step 2 above 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 $A_0, A_1, \ldots$ which are approximations to an unknown automata $A$, based on a sequence of observations of the input/output behaviour of $A$. The sequence $A_0, A_1, \ldots$ 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 two reasons.

(1) Real world systems are often too big to be completely learned and tested within a feasible timescale. This is mainly due to: (i) the time complexity of learning and model checking algorithms, and (ii) the time needed to execute the individual test cases on a large SUT.

(2) Testing of specific requirements such as use cases may not require learning and analysing the entire SUT $S$, but only the relevant fragment of $S$ which implements the requirement $Req$.

For these two reasons, the IKL learning algorithm used in LBTest is based on incremental learning.

This concept of a *relevant fragment* of an SUT for testing a requirement $Req$ raises the question of the relative efficiency of different types of queries (test cases). We have already seen that in LBT, test cases can be generated by model checking, by active learning, or by some other process entirely such as random querying.

As indicated in (1) above, the overhead of SUT execution time to answer an individual query can be large compared with the execution time of learning and model checking. There are examples of industrial systems where this execution time is of the order of minutes. So realistically, queries should be seen as “expensive”. From the viewpoint of relevance therefore, as many queries as possible should be derived from model checking the hypothesis automaton, since these queries are all based on checking the requirements $Req$. Conversely as few queries as possible should be derived from the active learning algorithm. Active learning queries have no way to reference the requirement $Req$, and therefore can only uncover an SUT error by accident. Furthermore, active learning queries may explore parts of the SUT which are irrelevant to checking $Req$, thereby leading the search for errors in a fruitless direction. Ideally, *every* query would represent a relevant and interesting requirements-based test case.

However, there is conflicting issue involved here, which is the computational effort needed to generate different types of queries. Model checker generated queries are generally computationally expensive relative to active learner generated queries, often by several orders of magnitude. Therefore, if too many (perhaps even all) queries are generated by model checking, then the LBT process may slow so much that random testing is simply faster. In a practical LBT tool, the ratio between the number of model checker generated queries, and the number of active learning queries must be controlled to achieve a balance between relevance and speed. The IKL algorithm implements a pragmatic balance
between these two types of queries that we have found to be reasonably efficient in practice.

Interestingly, when the balance of active learning queries becomes very high, and model checking queries are almost eliminated, we might think that LBT becomes similar to random testing. However [26] shows that this is not the case. Thus using active learner queries alone, LBT can achieve better functional coverage than random testing.

Projection

When we consider the output variables of the SUT that appear in a specific formal black box requirement \( Req \), we often see just a small subset of the set of all output variables of \( S \). This observation points to a powerful abstraction technique for learning that can be termed bit-slicing (for propositional variables) or more generally projection.

Like incremental learning, projection is another abstraction method that concentrates on learning only the relevant SUT behavior needed to test the requirement \( Req \). Essentially, projection involves learning a quotient model of the SUT by observing just the output variables appearing in \( Req \). Since quotient models of \( S \) may be dramatically smaller than \( S \) itself, the time needed for learning and testing may be considerably reduced. Therefore, projection seems to be an essential component of a scalable LBT system. Indeed, the combination of incremental learning and projection seems to be particularly powerful. The IKL algorithm incorporates both these features, and they will be discussed in further detail in Sections 3 and 4.

1.3 Literature Survey

Several previous works, (for example Peled et al. [22], Groce et al. [11] and Raffelt et al. [23]) 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. [7] and Chauhan et al. [6]). The LBT approach can be distinguished from these other approaches by: (i) an emphasis on testing rather than verification, and (ii) the use of incremental learning and other abstraction techniques specifically chosen to achieve scalable testing and faster error discovery (c.f. Section 1.2).

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 than incremental learning. Among the much smaller number of known incremental learning algorithms, we can mention the RPNII algorithm (Dupont [9]) and the IID algorithm (Parekh et al. [21]) which learn Moore automata, and the ICGE algorithm (Meinke and Fei [16]) which learns Mealy automata over abstract data types. No algorithm which combines incremental learning and projection has been published in the literature. The problem of integrating active learning queries with model checker generated queries (which in some sense take over the
role of Angluin’s equivalence checker [2]) has also not been considered. Thus:
(i) the design of the IKL algorithm, (ii) its formal proof of correctness, and
(iii) its motivation by efficient test case generation represent the main novel
contributions of our paper.

The use of minimisation algorithms in automata learning also seems not
to have been considered. This is mainly because most DFA learning algorithms
naturally infer the canonical minimal automaton. However, our use of projection
as an abstraction method for learning large Kripke structures does not lead
immediately to minimal structures. In fact, inferring non-minimal automata can
even lead to efficiency gains as we have shown elsewhere in [16].

For different automata models and different notions of equivalence, the com-
plexity of the minimisation problem can vary considerably. The survey [3] con-
siders minimisation algorithms for DFA up to language equivalence, with time
complexities varying between $O(n^2)$ and $O(n \log n)$. Kripke structures represent
a generalisation of DFA to allow non-determinism and multiple outputs. They
have been widely used to model concurrent and embedded systems. An algo-
rum for minimizing Kripke structures has been given in [5]. In the presence
of non-determinism, the complexity of minimisation is quite high. Minimisation
up to language equivalence requires exponential time, while minimisation up to
a weaker simulation equivalence can be carried out polynomial time (see [5]).
By contrast, we will show that deterministic Kripke structures can be efficiently
minimized even up to language equivalence with a worst case time complexity of
$O(\text{kn log}_{2} n)$. Our generalisation of Hopcroft’s DFA minimisation algorithm to
deterministic Kripke structures in Section 6 is fairly simple and straightforward.
Nevertheless, this algorithm has not been previously published in the literature,
and represents another novel contribution.

2 Mathematical Preliminaries and Notation

In this section we introduce some basic concepts and notations needed to define
and prove the correctness of the IKL learning algorithm. Let $\Sigma$ be any set of
symbols then $\Sigma^*$ denotes the set of all finite strings over $\Sigma$ including the empty
string $\varepsilon$. The length of a string $\alpha \in \Sigma^*$ is denoted by $|\alpha|$ and $|\varepsilon| = 0$. For strings
$\alpha, \beta \in \Sigma^*$, $\alpha \cdot \beta$ denotes their concatenation.

For $\alpha, \beta, \gamma \in \Sigma^*$, if $\alpha = \beta \gamma$ then $\beta$ is termed a prefix of $\alpha$ and $\gamma$ is termed
a suffix of $\alpha$. We let $\text{Pref}(\alpha)$ denote the prefix closure of $\alpha$, i.e. the set of all prefixes of $\alpha$. 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: $\Sigma$ 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 $\delta : Q \times \Sigma \rightarrow Q$ with the usual meaning, and can be inductively extended
to a mapping $\delta^* : Q \times \Sigma^* \to Q$ where $\delta^*(q, \varepsilon) = q$ and $\delta^*(q, \sigma_1, \ldots, \sigma_{n+1}) = \delta(\delta^*(q, \sigma_1, \ldots, \sigma_n), \sigma_{n+1})$.

A dead state is a state from which no accepting state can be reached, and a state which is not dead is termed live. Since input strings can be used to name states, given any 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 $f(\alpha, \sigma) = \alpha \cdot \sigma$ 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 allow 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

$$A = (Q_A, \Sigma, \delta_A : Q_A \times \Sigma \to Q_A, q_0^A, \lambda_A : Q_A \to \mathbb{B}^k)$$

where $Q_A$ is a state set, $\delta_A$ is the state transition function, $q_0^A$ is the initial state and $\lambda_A$ 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 $\lambda_A^*(\sigma_1, \ldots, \sigma_i) = \lambda_A(\delta_A^*(q, \sigma_1, \ldots, \sigma_i))$. Given any $R \subseteq Q$ we write $\lambda(R) = \cup_{r \in R} \lambda(r)$. We let $q, \sigma$ denote $\delta(q, \sigma)$ and $R, \sigma$ denotes $\{r, \sigma \mid r \in R\}$ for $R \subseteq Q$.

Note that a 1-bit deterministic Kripke structure $A$ is isomorphic to a DFA $A' = (Q_A', \Sigma, \delta_A' : Q_A' \times \Sigma \to Q_A', q_0^A, F_A')$, where $F_A' \subseteq Q_A'$ and $\lambda_A'(q) = true$ if, and only if $q \in F_A'$.

In the context of Boolean valued output variables, the concept of projection on a set of output variables will also be termed bit slicing. Let us make precise the concept of a bit-slice or projection of a Kripke structure.

2.2. Definition. Let $A$ be a $k$-bit Kripke structure over a finite input alphabet $\Sigma$,

$$A = (Q_A, \Sigma, \delta_A : Q_A \times \Sigma \to Q_A, q_0^A, \lambda_A : Q_A \to \mathbb{B}^k).$$

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

$$A_i = (Q_A, \Sigma, \delta_A : Q_A \times \Sigma \to Q_A, q_0^A, \lambda_A : Q_A \to \mathbb{B}),$$

and $\lambda_A(q) = \lambda_A(q)_i$, i.e., $\lambda_A(q)_i$ 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 a subdirect product construction. This will be discussed in Section 5.
A Kripke structure $\mathcal{A}$ is minimal if it has no proper subalgebra. This is equivalent to all states of $\mathcal{A}$ being reachable from the initial state by means of some input string. If $\mathcal{A}$ is a Kripke structure then $\mathcal{A}$ always has a minimal subalgebra which we denote by $\text{Min}(\mathcal{A})$.

### 3 Architecture of the IKL Algorithm

As discussed in Section 1, IKL is an algorithm for incrementally inferring a deterministic $k$-bit Kripke structure from observational data. For efficient testing, it also implements projection on output variables. An architectural view of the IKL algorithm is given in Figure 1. The basic idea of the algorithm is to learn a $k$-bit Kripke structure as a family of $k$ 1-bit Kripke structures (i.e. DFA) using an incremental DFA learning algorithm for each of the $k$ individual DFA.

![Architecture of the IKL Algorithm](image_url)

**Fig. 1.** Architecture of the IKL algorithm.

For DFA learning, we use an incremental refinement of Angluin’s ID algorithm [1]. The dead state $d_0$ used in the ID algorithm can be used in incremental learning as an abstraction for all currently unknown information about the system to be learned. Our refinement of ID differs from the IID learning algorithm described in [21] in ways which have been discussed in [24]. Note that the IKL architecture is modular, in the sense that other DFA learning algorithms could
be used instead of ID. These might alter the overall performance of IKL from a testing perspective (see Section 8).

As Figure 1 indicates, these $k$ incremental DFA learning algorithms must co-operate in order to jointly learn an entire family of DFA. This co-operation between the DFA learners is termed lazy learning. It is used to support more frequent model checking during testing, which is desirable for the reasons explained in Section 1.2. The goal of lazy learning then is to learn the DFA family in a way that can produce new $k$-bit hypothesis Kripke structures with maximum frequency.

The $k$ individual DFA are assembled into a single $k$-bit Kripke structure using a generalisation of the direct product construction known as a subdirect product. Without minimisation, the state space of the product automaton would be very large. However, the state space can be reduced on the fly, resulting in a subalgebra of the direct product. This removes all states which are not reachable from the initial state, using some input string. The state space of this subdirect product is typically still very large. In order to minimise the state space even further, we finally apply a minimisation algorithm for Kripke structures. For this we adapt Hopcroft’s minimisation algorithm for DFA [12], and generalise it to Kripke structures. We will discuss the state space sizes achieved by the intermediate Kripke structures during the incremental learning process in Section 7.

In the next three sections we define and prove correct the three major components of the IKL algorithm:

(i) the DFA family learning algorithm $FID$ (Section 4),
(ii) the subdirect product construction (Section 5), and
(iii) a Kripke structure minimisation algorithm (Section 6)

4 Incremental Learning of DFA Families

In this section we define and prove correct an algorithm $FID$ for incremental lazy learning of a family of DFA that share a common input. This approach supports bit-sliced learning of a large Kripke structure by projection of specific output variables (c.f. Section 1.2). Our algorithm is derived from the ID learning algorithm for DFA described in [1], and our correctness proof makes use of the correctness property of $ID$. Therefore, we begin by reviewing the ID algorithm itself, before turning our attention to DFA family learning.

4.1 The ID Algorithm

The $ID$ algorithm and its correctness have been discussed at length in [1]. Therefore our own presentation can be brief. 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.

A symbol $d_0$ is added as a name for the canonical dead state. Now this dead state can be used for incremental learning of a DFA, since parts of the DFA which have not yet been learned can be "hidden" inside the dead state. This is a key idea in the FID incremental learning algorithm described in Section 4.2.

The set of state names is then iteratively partitioned into sets $E_i(\alpha) \subseteq T'$ for $i = 0, 1, \ldots$ such that elements $\alpha, \beta$ 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 DFA as the resulting quotient DFA. 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 $V$.

In Section 4.2, we present the DFA family learning algorithm FID so that similar variables in the FID and ID algorithms share similar names. This pedagogic device emphasises some similarity in the behaviour of both algorithms. However, there are also important differences of behaviour. Thus, when analysing the behavioural properties of similar program variables, we will try to distinguish their context as $v^D_n$, $E^D_n(\alpha)$, etc, (for the ID algorithm) and correspondingly $v^*_n$, $E^*_n(\alpha)$, etc, (for the FID algorithm). Our basic argument in the proof of correctness of FID is to show how the learning behaviour of FID on a sequence of input strings $s_1, \ldots s_n \in \Sigma^*$ can be simulated by the behaviour of ID on the prefix closure $\text{Pref}(\{ s_1, \ldots s_n \})$ of the corresponding set of inputs $\{ s_1, \ldots s_n \}$. Once this is established one can apply the correctness of ID to establish the correctness of FID. The correctness of the ID algorithm can be stated as follows.

4.1.1. Theorem.

(i) Let $P \subseteq \Sigma^*$ be a live complete set for a DFA $A$ containing $\lambda$. Then given $P$ and $A$ as input, the ID algorithm terminates and the automaton $M$ returned is the canonical minimum state automaton for $L(A)$.

(ii) Let $l \in \mathbb{N}$ be the maximum value of program variable $i^D$ given $P$ and $A$. For all $0 \leq n \leq l$ and for all $\alpha \in T$,

\[ E^D_n(\alpha) = \{ \ v^D_j \in V^D \ | \ 0 \leq j \leq n, \ \alpha v^D_j \in L(A) \ \}. \]


(ii) By induction on $n$.

Basis. Suppose $n = 0$. Then $v^D_0 = \lambda$. For any $\alpha \in T$, if $\alpha v^D_0 \in L(A)$ then $\alpha \in L(A)$ so $E^D_0(\alpha) = \{ v^D_0 \}$. If $\alpha v^D_0 \notin L(A)$ then $\alpha \notin L(A)$ so $E^D_0(\alpha) = \emptyset$. Thus $E^D_0(\alpha) = \{ v^D_j \ | \ 0 \leq j \leq 0, \ \alpha v^D_j \in L(A) \ \}$.

Induction Step. Suppose $l \geq n > 0$. Consider any $\alpha, \beta \in P'$ and $b \in \Sigma$ such that $E^D_{n-1}(\alpha) = E^D_{n-1}(\beta)$ but $E^D_{n-1}(f(\alpha, b)) \neq E^D_{n-1}(f(\beta, b))$. Since $n - 1 < l$
Algorithm 1 ID Learning Algorithm

**Input:** A live complete set $P \subseteq \Sigma^*$ and a DFA $A$ to act as a teacher answering membership queries $\alpha \in L(A)$?

**Output:** A DFA $M$ language equivalent to the target DFA $A$.

1. begin
2. //Perform Initialization
3. $i = 0$, $v_i = \lambda$, $V = \{ v_i \}$,
4. $T = P \cup \{ f(\alpha, b) | (\alpha, b) \in P \times \Sigma \}$, $T' = T \cup \{ d_0 \}$
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.
then $\alpha$, $\beta$ 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 $E_n^{ID}(\alpha) = \{ v_j^{ID} \in V^{ID} \mid 0 \leq j \leq n, \ \alpha v_j^{ID} \in L(A) \}$.

4.2 The FID Algorithm

We can now present the FID algorithm for incremental lazy learning of a family of DFA. We give a rigorous proof that FID correctly learns in the limit in the sense of [10] (Correctness Theorem 4.2.6).

Algorithm 2 is the main component of the FID algorithm. It learns a sequence $F_0, \ldots, F_l$ of families $F_i = (M_1^i, \ldots, M_n^i)$ of $n$ DFA driven by a sequence of input strings (queries) $s_1, \ldots, s_l$. The teacher is a single $n$-bit Kripke structure $A$. Then $F_0$ is a null hypothesis about the projections $A_1, \ldots, A_n$ of $A$. We claim that the sequence $F_0, F_1, \ldots$ finitely converges to the projections $(A_1, \ldots, A_n)$ given enough information about $A$, i.e. when $s_1, \ldots, s_l$ contains a live complete set of queries for each projection $A_i$.

The basic idea of Algorithm 2 is to construct in parallel a family
\[ (E_{i_1}^1, \ldots, E_{i_n}^n) \]
of $n$ individual equivalence relations on the same set $T_k$ of state names. For each equivalence relation $E_{i_j}^j$, a set $V_j$ of distinguishing strings is incrementally generated to split pairs of equivalence classes in $E_{i_j}^j$ until a congruence is achieved. Then a quotient DFA $M^j$ can be constructed from the partition of $T_k$ by the congruence $E_{i_j}^j$. The congruences are constructed so that $E_{i_j}^j \subseteq E_{i_{j+1}}^j$ and thus the FID algorithm is incremental, and fully reuses information about previous approximations, which is efficient.

Each DFA family $F_l = (M_1^1, \ldots, M_n^l)$ is constructed from the partition family $(E_{i_1}^1, \ldots, E_{i_n}^n)$ using Synthesis Algorithm 4. When the FID algorithm is applied to the problem of LBT, the input strings $s_i \in \Sigma^*$ to FID are generated as counterexamples to correctness (i.e. test cases). For this we execute a model checker on a Kripke structure $A_{i-1}$ which is a minimised subdirect product of $(M_{i-1}^1, \ldots, M_{i-1}^n)$ using a requirements specification $\phi$ expressed in temporal logic. (The construction of $A_{i-1}$ will be detailed in Sections 5 and 6.) In the case that no counterexamples to $\phi$ can be found in $A_{i-1}$ then $s_i$ is randomly chosen, taking care to avoid all previously used input strings.

Algorithm 3 implements lazy partition refinement, to extend $E_{i_1}^1, \ldots, E_{i_n}^n$ from being equivalence relations on states to being a family of congruences with respect to the state transition functions $\delta_1, \ldots, \delta_n$ for the synthesized DFA $M_1^1, \ldots, M_n^l$.

Thus line 1 in Algorithm 3 searches for congruence failure in any one of the equivalence relations $E_{i_1}^1, \ldots, E_{i_n}^n$. In lines 6-14 of Algorithm 3 we apply
Algorithm 2 FID: a DFA Family Learning Algorithm

Input: A file $S = s_1, \ldots, s_l$ of input strings $s_i \in \Sigma^*$ and an $n$-bit Kripke structure $\mathcal{A}$ as teacher to answer queries $\lambda_A^* (s_i) = ?$

Output: A sequence of families $F_t = (M^1_t, \ldots, M^n_t)$ of DFA for $t = 0, \ldots, l$.

1. begin
2. //Perform Initialization
3. for $c = 1$ to $n$ do \{ $i_c = 0, v^i_c = \varepsilon, V_c = \{ v^c_c \}$ \}
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. $\forall \alpha \in T_0 \{ \theta, \ldots, b_n = \lambda_A^* (\alpha) \}$
10. for $c = 1$ to $n$ do \{ $E^0_c (\alpha) = \{ v^c_c \}$ else $E^0_c (\alpha) = \emptyset$ \}
11. //Refine the initial equivalence relations $E^1_0, \ldots, E^n_0$
12. //into congruences using Algorithm 3
13. //Synthesize an initial family $F_0$ approximating $A$
15. //Process the file of examples.
16. while $S \neq$ empty do \{
17. read( S, $\alpha$ )
18. $k = k+1, t = t+1$
19. $P_k = P_{k-1} \cup \text{Pref} (\alpha)$ //prefix closure
20. $P'_k = P_k \cup \{ d_0 \}$
21. $T_k = P_k \cup \{ f(\alpha, b) \mid \alpha \in P_k - P_{k-1}, b \in \Sigma \}$ //for prefix closure
22. $T'_k = T_k \cup \{ d_0 \}$
23. $\forall \alpha \in T_k - T_{k-1}$ \{
24. for $c = 1$ to $n$ do $E^0_c (\alpha) = \emptyset$ //initialise the new equivalence class $E^0_0 (\alpha)$
25. for $j = 0$ to $i_c$ do \{
26. // Consider adding previous distinguishing string $v^j_c \in V_c$
27. // to the new equivalence class $E^j_c (\alpha)$
28. $(b_1, \ldots, b_n) = \lambda_A^* (\alpha \cdot v^j_c)$
29. if $b_c$ then $E^j_c (\alpha) = E^j_c (\alpha) \cup \{ v^c_c \}$
30. \}
31. //Refine the current equivalence relations $E^1_{i_1}, \ldots, E^n_{i_n}$
32. // into congruences using Algorithm 3
33. if $\alpha$ is consistent with $F_{t-1}$
34. then $F_t = F_{t-1}$
35. else synthesize the family $F_t$ using Algorithm 4.
36. \}
37. end.
Algorithm 3 Lazy Partition Refinement

1. while (\(\exists 1 \leq c \leq n, \exists \alpha, \beta \in P_k\) and \(\exists \sigma \in \Sigma\) such that \(E_{i_c}^c(\alpha) = E_{i_c}^c(\beta)\) but \(E_{i_c}^c(f(\alpha, \sigma)) \neq E_{i_c}^c(f(\beta, \sigma))\)) do {

2. //Equivalence relation \(E_{i_c}^c\) is not a congruence w.r.t. \(\delta_c\)
3. //so add a new distinguishing sequence.
4. Choose \(\gamma \in E_{i_c}^c(f(\alpha, \sigma)) \oplus E_{i_c}^c(f(\beta, \sigma))\)
5. \(v = \sigma \cdot \gamma\)
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 \(E_{i_c}^c(\alpha) = E_{i_c}^c(\beta)\) and \(E_{i_c}^c(f(\alpha, \sigma)) \neq E_{i_c}^c(f(\beta, \sigma))\) then {
10. //Lazy refinement of equivalence relation \(E_{i_c}^c\)
11. \(i_c = i_c + 1, v_i = v, V_c = V_c \cup \{v_i\}\)
12. if \(b_c\) then \(E_{i_c}^c(\alpha) = E_{i_c}^c(\alpha) \cup \{v_i\}\) else \(E_{i_c}^c(\alpha) = E_{i_c+1}^c(\alpha)\)
13. }
14. }
15. }

Lazy partition refinement. This technique implies reusing the new distinguishing string \(v\) wherever possible to refine each equivalence relation \(E_{i_j}^j\) that is not yet a congruence. On the other hand, any equivalence relation \(E_{i_j}^j\) that is already a congruence is not refined, even though the result \(b_j\) of the new query \(\alpha \cdot v\) might add some new information to \(M^j\). This brings the set of relations \(E_{i_1}^1, \ldots, E_{i_n}^n\) to a simultaneous fixed point of \(n\) congruence constructions as soon as possible. It therefore helps to reduce the number of active learner queries and raise the number of model checker queries used during learning based testing (cf. Section 1.2).

We begin an analysis of the correctness of the FID algorithm by confirming that the construction of hypothesis DFA carried out by Algorithm 4 is well defined.

4.2.1. Proposition. For each \(t \geq 0\) the hypothesis DFA \(M^1, \ldots, M^n\) constructed by the DFA Family Synthesis Algorithm 4 after \(t\) input strings have been applied to \(A\) are all well defined DFA.

Proof. The main task is to show \(\delta\) to be well defined function and uniquely defined for every state \(E_i(\alpha)\), where \(\alpha \in T_k\).

Proposition 4.2.1 establishes that Algorithm 4 will generate families of well defined DFA. However, to show that the FID algorithm learns correctly in the limit, we must prove that this sequence of DFA families finitely converges to the \(n\) individual projections \(A_i\) of the target Kripke structure \(A\). It will suffice to show that the behaviour of FID 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 4.4.1.(i)). The first step in this proof is to show that the sequences of
Algorithm 4 DFA Family Synthesis

1. for c = 1 to n do 
2.   // Synthesize the quotient DFA $M^c$
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 $\delta_c$ of $M^c$ is defined as follows:
7.     $\forall \alpha \in P^c_k$ 
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 $\forall b \in \Sigma$ \{ $\delta_c(E^c_{i_c}(\alpha), b) = E^c_{i_c}(\alpha \cdot b)$ \}
10.   $\forall \beta \in T_k - P^c_k$ 
11.   if $\forall \alpha \in P^c_k$ \{ $E^c_{i_c}(\beta) \neq E^c_{i_c}(\alpha)$ \} and $E^c_{i_c}(\beta) \neq \emptyset$ then
12.     $\forall b \in \Sigma$ \{ $\delta_c(E^c_{i_c}(\beta), b) = \emptyset$ \}
13.   }
14.   }
15.   return $F = (M^1, \ldots, M^n)$

sets of state names $P_k$ and $T_k$ generated by FID converge to the sets $P^{ID}$ and $T^{ID}$ of ID.

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

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

Proof. Clearly (iii) follows from (i) and (ii). Then (i) and (ii) are easily proved by induction on $k$.

Observe that unlike FID, the ID algorithm does not compute any prefix closure of input strings. Therefore, prefix closure must be added explicitly in Proposition 4.2.2, to make a correspondence between the behaviour of FID and ID.

Next we turn our attention to proving some fundamental loop invariants for Algorithm 2. Since this algorithm in turn calls the Lazy Partition Refinement Algorithm 3 then we have in effect a doubly nested loop structure to analyse. Clearly the outer loop counter $k$ in Algorithm 2 and the family of inner loop counters $i_c$ (for $1 \leq c \leq n$) in Algorithm 3 both increase on each iteration. However, the relationships between these counter variables are not easily defined. Nevertheless, since all variables increase from an initial value of zero, we can assume the existence of some family of $n$ monotone re-indexing functions that capture their relationships.
4.2.3. Definition. Let \( S = s_1, \ldots, s_l \) be any non-empty sequence of strings \( s_i \in \Sigma^* \). The re-indexing function \( K^S_c : \mathbb{N} \to \mathbb{N} \) for FID on input \( S \) (for each \( 1 \leq c \leq n \)) is the unique monotonically increasing function such that for each \( n \in \mathbb{N} \), \( K^S_c(n) \) is the least integer \( m \) such that program variable \( k \) has value \( m \) while the program variable \( i_c \) has value \( n \). Thus, for example, \( K^S_c(0) = 0 \) for all \( 1 \leq c \leq n \). When \( S \) is clear from the context, we may simply write \( K_c \) for \( K^S_c \).

With the help of these re-indexing functions we can express important invariant properties of the distinguishing sequence variables \( v^j_c \) and partition set variables \( E^*_c(\alpha) \). Using Proposition 4.2.2 their relationship to the corresponding variables \( v^{ID}_j \) and \( E^{ID}_n(\alpha) \) of ID can be established. Since Algorithm 2 has a doubly nested loop structure, the proof of Simulation Theorem 4.2.4 below makes use of a doubly nested induction argument.

4.2.4. Simulation Theorem. Let \( S = s_1, \ldots, s_l \) be any non-empty sequence of strings \( s_i \in \Sigma^* \). For any execution of FID on \( S \) and the \( n \)-bit Kripke structure \( \mathcal{A} \) there exists an execution of ID on \( \text{Pref}(\{ \lambda, s_1, \ldots, s_l \}) \) and the \( c \)-th projection \( \mathcal{A}_c \) (for each \( 1 \leq c \leq n \)) such that for all \( m \geq 0 \):

(i) For all \( n \geq 0 \), if \( K_c(n) = m \) then:
   
   (a) for all \( 0 \leq j \leq n \), \( v^j_c = v^{ID}_j \),
   
   (b) for all \( 0 \leq j < n \), \( v^n_c = v^{ID}_j \),
   
   (c) for all \( \alpha \in T_m \), \( E^*_c(\alpha) = \{ v^j_c \in V_c \mid 0 \leq j \leq n, \alpha v^j_c \in \mathcal{L}(\mathcal{A}_c) \} \).

(ii) If \( m > 0 \) then let \( p \in \mathbb{N} \) be the greatest integer such that \( K_c(p) = m - 1 \). Then for all \( \alpha \in T_m \), \( E^*_p(\alpha) = \{ v^j_c \in V_c \mid 0 \leq j \leq p, \alpha v^j_c \in \mathcal{L}(\mathcal{A}_c) \} \).

(iii) The \( m \)-th partition refinement of FID terminates.

Proof. By induction on \( m \) using Proposition 4.2.2.(i).

Part (i.a) above asserts that the same distinguishing sequences are produced in the same order by FID and ID. Part (i.b) asserts that a distinguishing sequence is never produced twice by FID. Part (i.c) and (ii) characterise the partition sets \( E^*_c(\alpha) \) as sets of all distinguishing sequences \( v^j_c \) that lead to an accepting state of \( \mathcal{A}_c \) from \( \alpha \).

Note that both ID and FID are non-deterministic algorithms (due to the non-deterministic choice on line 17 of Algorithm 1 and line 4 of Algorithm 3). Therefore in the statement of Theorem 4.2.4 above, we can only talk about the existence of some correct simulation. Clearly there are also simulations of FID by ID which are not correct, but this does not affect the basic correctness argument.

4.2.5. Corollary. Let \( S = s_1, \ldots, s_l \) be any non-empty sequence of strings \( s_i \in \Sigma^* \). Any execution of FID on \( S \) and an \( n \)-bit Kripke structure \( \mathcal{A} \) terminates with the program variable \( k \) having value \( l \).

Proof. Follows from Simulation Theorem 4.2.4.(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 \( P_k \) and \( T_k \) in Proposition 4.2.2 and \( v^j_c \) and \( E^*_n(\alpha) \) in Simulation Theorem
4.2.4 it is now a simple matter to establish correctness of learning for the FID Algorithm.

4.2.6. Correctness Theorem. Let \( S = s_1, \ldots, s_t \) be any non-empty sequence of strings \( s_i \in \Sigma^* \) such that \( \{ \lambda, s_1, \ldots, s_t \} \) contains a live complete set for each projection \( A_i \) of \( A \). Then FID terminates on \( S \). Also for each \( 1 \leq i \leq n \) the hypothesis DFA \( M^i_t \) is a canonical representation of \( A_i \).

**Proof.** By Corollary 4.2.5, FID terminates on \( S \) with the variable \( k \) having value \( l \). By Simulation Theorem 4.2.4.(i) and Theorem 4.1.1.(ii), there exists an execution of ID on \( \text{Pref} \{ \lambda, s_1, \ldots, s_t \} \) such that \( E^i_t(\alpha) = E^{ID}_i(\alpha) \) for all \( \alpha \in T_t \) and any \( n \) such that \( K(n) = l \). By Proposition 4.2.2.(iii), \( T_t = T^{ID}_t \) and \( P'_t = P' \). So letting \( M^{ID}_t \) be the canonical representation of \( A_i \) constructed by ID using \( \text{Pref} \{ \lambda, s_1, \ldots, s_t \} \) then \( M^{ID}_t \) and \( M^i_t \) have the same state sets, initial states, accepting states and transitions.

Our next result confirms that each hypothesis DFA \( M^i_t \) generated after \( t \) input strings have been applied to \( A \) is consistent with all currently known observations about the \( i \)th projection \( A_i \). This is quite straightforward in the light of Simulation Theorem 4.2.4.

4.2.7. Compatibility Theorem. Let \( S = s_1, \ldots, s_t \) 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 \( M^i_t \) accepts \( s \) if, and only if the \( i \)th projection \( A_i \) of \( A \) does.

**Proof.** By definition, \( M^i_t \) is compatible with \( A_i \) on \( \{ \lambda, s_1, \ldots, s_t \} \) if, and only if, for each \( 0 \leq j \leq t \), \( s_j \in L(A_i) \) \( \Leftrightarrow \lambda \in E^i_t(s_j) \), where \( i_t \) is the greatest integer such that \( K(i_t) = t \) and the sets \( E^i_t(\alpha) \) for \( \alpha \in T_t \) are the states of \( M^i_t \). Now \( v_0 = \lambda \). So by Simulation Theorem 4.2.4.(i),(c), if \( s_j \in L(A_i) \) then \( s_j, v_0 \in L(A_i) \) so \( v_0 \in E^i_t(s_j) \), i.e. \( \lambda \in E^i_t(s_j) \), and if \( s_j \not\in L(A_i) \) then \( s_j, v_0 \not\in L(A_i) \) so \( v_0 \not\in E^i_t(s_j) \), i.e. \( \lambda \not\in E^i_t(s_j) \).

We have now established a reliable method for decomposing the problem of learning a \( k \)-bit Kripke structure \( A \) into the problem of learning a family of \( k \) individual DFA. This approach supports projection, as defined in Section 1.2 and Definition 2.2.

5 Subdirect Product Construction

We next turn our attention to problem of efficiently recombining a family of \( k \) individual DFA (the projections) into a single \( k \)-bit deterministic Kripke structure. For this we use a well known algebraic construction known as a subdirect product. Informally, a subdirect product of a family \( F = \{ A_i \mid i \in I \} \) of algebraic structures, is any subalgebra of the direct product \( \Pi F = \Pi_{i \in I} A_i \) which projects onto (surjectively) each of its co-ordinate algebras \( A_i \). The subdirect product construction was introduced in [4] as a universal decomposition method applicable to any algebraic structures. The reader may consult [19] for basic facts.
about subdirect products and their universal properties. A specific definition for
deterministic Kripke structures is given below.

To begin with, we observe that for black-box testing it suffices to learn a
Kripke structure up to behavioural equivalence.

5.1. Definition. Let $A$ and $B$ be $k$-bit Kripke structures over a finite input
alphabet $\Sigma$. We say that $A$ and $B$ are behaviorally 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 struc-
tures and DFA, for such structures we have $A \equiv B$ if, and only if, $L(A') = L(B')$. Furthermore, if $\text{Min}(A)$ is the minimal subalgebra of $A$ then $\text{Min}(A) \equiv A$.

A family of $k$ individual 1-bit Kripke structures (DFA) can be combined
into a single $k$-bit Kripke structure using the following instance of the subdirect
product construction.

5.2. Definition. Let $A_1, \ldots, A_k$ be a family of 1-bit Kripke structures,$\quad A_i = (Q_i, \Sigma, \delta_i : Q_i \times \Sigma \to Q_i, q_0^i, \lambda_i : Q \to B_i)$
for $i = 1, \ldots, k$. Define the direct product Kripke structure

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

where $Q = \prod_{i=1}^k Q_i = Q_1 \times \ldots \times Q_k$ and $q^0 = (q_0^1, \ldots, q_0^k)$. Also

$$\delta(q_1, \ldots, q_k, \sigma) = (\delta_1(q_1, \sigma), \ldots, \delta_k(q_k, \sigma)), \quad \lambda(q_1, \ldots, q_k) = (\lambda_1(q_1), \ldots, \lambda_k(q_k)).$$

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

$$\text{proj}_i : Q_1 \times \ldots \times Q_k \to Q_i, \quad \text{proj}_i(q_1, \ldots, q_k) = q_i, \quad 1 \leq i \leq k$$

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

The reason for taking the subdirect product of the $A_i$ as the minimal sub-
algebra 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 requirements testing they are irrelevant. This subdirect product can be
computed from its components $A_i$ in time $O(k.m.|\Sigma|)$ where $m$ is the number of
states in the resulting subdirect product and $|\Sigma|$ is the size of the input alphabet. A naive algorithm based on systematic path exploration starting from the
initial state can be used. We leave the definition of this algorithm as an exercise for the reader.

As is well known from universal algebra, the \( i \)-th projection mapping \( \text{proj}_i \) is a homomorphism.

5.3. Proposition. Let \( A_1, \ldots, A_k \) be any minimal 1-bit Kripke structures.

(i) For each \( 1 \leq i \leq k \), the projection mapping \( \text{proj}_i : \text{Min}( \prod_{i=1}^{k} A_i ) \rightarrow A_i \) is an epimorphism. Hence \( \text{Min}( \prod_{i=1}^{k} A_i ) \) is a subdirect product of the \( A_i \).

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

Proof. (i) Immediate since the \( A_i \) are minimal. (ii) Follows from the fact that \( \text{Min}(A) \equiv A \).

The following theorem justifies bit-sliced learning of \( k \)-bit Kripke structures using conventional regular inference methods for a family of DFA. It constitutes the correctness argument for the subdirect product component of the IKL architecture, as presented in Section 3.

5.4. Theorem. Let \( A \) be a \( k \)-bit Kripke structure over a finite input alphabet \( \Sigma \). 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
\[
A \equiv \text{Min}( \prod_{i=1}^{k} B_i )
\]

Proof. Use Proposition 5.3.

By Correctness Theorem 4.2.6, the assumptions of Theorem 5.4 on the 1-bit Kripke structures \( B_1, \ldots, B_k \) are fulfilled by the IKL architecture, since these are the canonical representations of \( A_1, \ldots, A_k \). So by Theorem 5.4, the output of the IKL algorithm, after DFA family learning has converged and the subdirect product construction has been applied is a \( k \)-bit Kripke structure \( B \) that is behaviourally equivalent with the input Kripke structure \( A \).

Despite the canonical DFA \( B_1, \ldots, B_k \) being minimal, the reduced product \( B \) may still be much larger in state space size than \( A \). This can slow down the process of model checking the output of IKL considerably. So it is important to reduce the state space size of \( B \) even further. This last step of the IKL algorithm will be discussed in the next section.

6 Kripke Structure Minimisation.

In this section we introduce an efficient algorithm for the minimisation of deterministic Kripke structures with \( O(|\Sigma|n \log_2 n) \) time complexity. Here \( n \) is the state space size of the Kripke structure \( A \) and \( |\Sigma| \) is the size of its input alphabet. This algorithm is applied on the back end of the IKL learning algorithm in order to speed up model checking of the learned hypothesis automata during testing.
To define a minimisation algorithm, we need to generalise the concepts of right language and Nerode congruence from DFA to deterministic Kripke structures. We then show how Hopcroft’s DFA minimisation algorithm of [12] can be generalised to compute the Nerode congruence \( \equiv \) of a deterministic Kripke structure \( A \). The quotient Kripke structure \( A/ \equiv \) is minimal and language equivalent to \( A \). This fact is the final result needed to prove the correctness of the IKL architecture. We will prove the correctness and complexity properties of our minimisation algorithm from first principles.

6.1 Minimal Deterministic Kripke Structures

Let us consider a DFA \( A = (Q, \Sigma, \delta, q_0, F) \). For each state \( q \in Q \) of \( A \) there corresponds a subautomaton of \( A \) rooted at \( q \) which accepts the regular language \( \mathcal{L}_q(A) \subseteq \Sigma^* \), consisting of just those words accepted by the subautomaton with \( q \) as initial state. Thus \( \mathcal{L}_{q_0}(A) \) is the language accepted by \( A \). The language \( \mathcal{L}_q(A) \) is called either the future of state \( q \) or the right language of \( q \). \( A \) is minimal (i.e. state minimal as opposed to algebraically minimal) i f for each pair of distinct states \( p, q \in Q \), we have, \( \mathcal{L}_p(A) \neq \mathcal{L}_q(A) \). For any regular language \( \mathcal{L} \subseteq \Sigma^* \) there is a smallest DFA (in terms of the number of states) accepting \( \mathcal{L} \). This DFA is minimal, and is unique up to isomorphism.

An equivalence relation \( \equiv \) can be defined on the states of a DFA by \( p \equiv q \) if and only if \( \mathcal{L}_p(A) = \mathcal{L}_q(A) \). This relation is a congruence, i.e. if \( p \equiv q \) then \( p.\sigma \equiv q.\sigma \) for all \( \sigma \in \Sigma^* \). It is known as the Nerode congruence. Consider the quotient DFA \( A/ \equiv \). This is the unique smallest DFA which accepts the regular language \( \mathcal{L}_{q_0}(A) \). The problem of minimizing a DFA \( A \) is therefore to compute its Nerode congruence, which will be the identity relation if, and only if \( A \) is a minimal automaton.

The problem of computing a minimal Kripke structure \( A \) is an analogous but more general problem. In this case, the right language \( \mathcal{L}_q(A) \) associated with a state \( q \) of \( A \) can be defined by

\[
\mathcal{L}_q(A) = \{ (\sigma_1, ..., \sigma_n, a) \in \Sigma^* \times \mathbb{B}^k \mid \lambda_q^*(\sigma_1, ..., \sigma_n) = a \}.
\]

As before, \( A \) is minimal if for each pair of distinct states \( p, q \in Q \) we have, \( \mathcal{L}_p(A) \neq \mathcal{L}_q(A) \). There is again a smallest Kripke structure associated with a right language \( \mathcal{L} \subseteq \Sigma^* \times \mathbb{B}^k \). This Kripke structure is also minimal, and unique up to isomorphism. The Nerode congruence for a Kripke structure \( A \) is now defined by:

\[
p \equiv q \text{ if and only if } \lambda_p^*(\sigma_1, ..., \sigma_n) = \lambda_q^*(\sigma_1, ..., \sigma_n) \text{ for all } (\sigma_1, ..., \sigma_n) \in \Sigma^*.
\]

and \( A/ \equiv \) is the unique smallest Kripke structure associated with the right language \( \mathcal{L}_{q_0}(A) \). So the problem of minimising \( A \) is to compute this congruence.

6.2 A Kripke Structure Minimisation Algorithm

Algorithm 5 presents an efficient algorithm to compute the Nerode congruence \( \equiv \) of a deterministic Kripke structure \( A \), which is the same as the state set
Algorithm 5 Kripke Structure Minimisation

Input: A deterministic Kripke structure $\mathcal{A}$ with no unreachable states and $k$ output bits.

Output: The Nerode congruence $\equiv$ for $\mathcal{A}$, i.e. equivalence classes of states for
the minimized structure $\mathcal{A}_{\text{min}}$ behaviourally equivalent to $\mathcal{A}$.

1 Create an initial state partition $P = \{B_q = \{q' \in Q \mid \lambda(q) = \lambda(q')\} \mid q \in Q\}$. Let $n = |P|$. Let $B_1, \ldots, B_n$ be an enumeration of $P$.

2 if $n = |Q|$ then go to line 30.

3 foreach $\sigma \in \Sigma$ do

4 for $i \leftarrow 1$ to $n$ do

5 $B(\sigma, i) = \{q \in B_i \mid \exists r \in Q \ s.t \ \delta(r, \sigma) = q\}$. /*This constitutes the subset
of states in block $B_i$ which have predecessors through input $\sigma$. */

6 $\text{count} = n + 1$;

7 foreach $\sigma \in \Sigma$ do

8 choose all the subsets $B(\sigma, i)$ (excluding any empty subsets) and put their
block numbers $i$ on a waiting list (i.e. an unordered set) $W(\sigma)$ to be processed.

9 Boolean splittable = true;

10 while splittable do

11 foreach $\sigma \in \Sigma$ do

12 foreach $i \in W(\sigma)$ do

13 Delete $i$ from $W(\sigma)$

14 for $j \leftarrow 1$ to $\text{count} - 1$ s.t. $\exists t \in B_j$ with $\delta(t, \sigma) \in B(\sigma, i)$ do

15 Create $B_j' = \{t \in B_j \mid \delta(t, \sigma) \in B(\sigma, i)\}$

16 if $B_j' \subset B_j$ then

17 $\hat{B}_{\text{count}} = B_j - B_j'$; $B_j = B_j'$

18 foreach $\sigma \in \Sigma$ do

19 $B(\sigma, \text{count}) = \{q \in B(\sigma, j) \mid q \in \hat{B}_{\text{count}}\}$;

20 $B(\sigma, j) = \{q \in B(\sigma, j) \mid q \in B_j\}$

21 if $j \notin W(\sigma)$ and $0 < |B(\sigma, j)| \leq |B(\sigma, \text{count})|$ then

22 $W(\sigma) = W(\sigma) \cup \{j\}$

23 else

24 $W(\sigma) = W(\sigma) \cup \{\text{count}\}$

25 $\text{count} = \text{count} + 1$;

26 splittable = false;

27 foreach $\sigma \in \Sigma$ do

28 if $W(\sigma) \neq \emptyset$ then

29 splittable = true;

30 Return partition blocks $B_1, \ldots, B_{\text{count}}$. 
of the associated quotient Kripke structure \( A/\equiv \). We will give a rigorous but simple proof of the correctness of this algorithm. By means of a new induction argument, we have simplified the correctness argument compared with [3] and [12]. First let us establish termination of the algorithm by using an appropriate well-founded ordering for the main loop variant.

6.2.1. Definition. Consider any pair of finite sets of finite sets \( A = \{A_1, \ldots, A_m\} \) and \( B = \{B_1, \ldots, B_n\} \). We define an ordering relation \( \leq \) on \( A \) and \( B \) by \( A \leq B \) if \( \forall 1 \leq i \leq m, \exists 1 \leq j \leq n \) such that \( A_i \subseteq B_j \). Define \( A < B \iff A \leq B \& A \neq B \). Clearly \( \leq \) is a reflexive, transitive relation. Furthermore \( \leq \) is well-founded, i.e. there are no infinite descending chains \( A_1 > A_2 > A_3 \ldots \), since \( \emptyset \) is the smallest element under \( \leq \).

6.2.2. Proposition. Algorithm 5 always terminates.

Proof. We have two cases for the termination of the algorithm as a result of the partition formed on line 1 of the algorithm: (1) when \( n = |Q| \), and (2) when \( n < |Q| \).

Consider the case when \( n = |Q| \) then each block in the partition corresponds to a state of the given Kripke structure with a unique bit-label and hence in this case the algorithm will terminate on line 30 by providing the description of these blocks.

Now consider the case when \( n < |Q| \). Then the waiting sets \( W(\sigma) \) for all \( \sigma \in \Sigma \) will be initialized on lines 7, 8 and the termination of the algorithm depends on proving the termination of the loop on line 10. Now \( W(\sigma) \) is initialized by loading the block numbers of the split sets on line 8. There are only two possibilities after any execution of the loop. Let \( W_m(\sigma) \) and \( W_m+1(\sigma) \) represent the state of the variable \( W(\sigma) \) before and after one execution of the loop respectively at any given time. Then either \( W_m(\sigma) = W_m+1(\sigma) \cup \{i\} \) and no splitting has taken place and \( i \) is the deleted block number, or \( W_m(\sigma) \cup \{j\} = W_m+1(\sigma) \cup \{i\} \) or \( W_m(\sigma) \cup \{k\} = W_m+1(\sigma) \cup \{i\} \) where \( j \) and \( k \) represent the split blocks and one of them goes into \( W_m(\sigma) \) if it has fewer incoming transitions. In either case \( W_m(\sigma) > W_m+1(\sigma) \) by Definition 6.2. Therefore \( W(\sigma) \) strictly decreases with each iteration of the loop on line 10. Since the ordering \( \leq \) is well-founded, Algorithm 5 must terminate.

Now we only need to show that when Algorithm 5 has terminated, it returns the Nerode congruence \( \equiv \) on states.

6.2.3. Proposition. Let \( P_i \) be the partition (block set) on the \( i \)th iteration of Algorithm 5. For any blocks \( B_j, B_k \in P_i \) and any states \( p \in B_j, q \in B_k \) if \( j \neq k \) then \( p \neq q \).

Proof. By induction on the number \( i \) of times the loop on line 10 is executed.

Basis: Suppose \( i = 0 \) then clearly the result holds because each block created at line 1 is distinguishable by the empty string \( \epsilon \).

Induction Step: Suppose \( i = m > 0 \). Let us assume that the proposition holds after \( m \) executions of the loop.
Consider any \( B_j, B_k \in P_\mathcal{m} \). During the \( m + 1 \)th execution of the loop on line 10 either block \( B_j \) is split into \( B'_j \) and \( B''_j \) or \( B_k \) is split into \( B'_k \) and \( B''_k \) but not both during one execution of the loop (due to line 17).

Consider the case when \( B_j \) is split then for any \( p \in B_j \), either \( p \in B'_j \) or \( p \in B''_j \). But for any \( p \in B_j \) and \( q \in B_k \), \( p \not\equiv q \) by the induction hypothesis. Therefore, for \( p \in B'_j \) or \( p \in B''_j \) \( p \not\equiv q \). Hence the proposition is true for \( m + 1 \)th execution of the loop in this case.

By symmetry the same argument holds when \( B_k \) is split.

The following Lemma gives a simple, but very effective way to understand Algorithm 5. Note that this analysis is more like a temporal logic argument than a loop invariant approach. This approach reflects the non-determinism inherent in the algorithm.

**6.2.4. Lemma.** For any states \( p, q \in Q \), if \( p \not\equiv q \) and initially \( p \) and \( q \) are in the same block \( p, q \in B_{i_0} \) then eventually \( p \) and \( q \) are split into different blocks, \( p \in B_j \) and \( q \in B_k \) for \( j \not= k \).

**Proof.** Suppose that \( p \not\equiv q \) and that initially \( p, q \in B_{i_0} \) for some block \( B_{i_0} \). Since \( p \not\equiv q \) then for some \( n \geq 0 \), and \( \sigma_1, \ldots, \sigma_n \in \Sigma \),

\[
\lambda^*(p, \sigma_1, \ldots, \sigma_n) \neq \lambda^*(q, \sigma_1, \ldots, \sigma_n).
\]

We prove the result by induction on \( n \).

**Basis** Suppose \( n = 0 \), so that \( \lambda(p) \neq \lambda(q) \). By line 1, \( p \in B_p \) and \( q \in B_q \) and \( B_p \neq B_q \). So the implication holds vacuously.

**Induction Step** Suppose \( n > 0 \) and for some \( \sigma_1, \ldots, \sigma_n \in \Sigma \),

\[
\lambda^*(p, \sigma_1, \ldots, \sigma_n) \neq \lambda^*(q, \sigma_1, \ldots, \sigma_n).
\]

(a) Suppose initially \( \delta(p, \sigma_1) \in B(\sigma_1, \alpha) \) and \( \delta(q, \sigma_1) \in B(\sigma_1, \beta) \) for \( \alpha \neq \beta \).

Consider when \( \sigma = \sigma_1 \) on the first iteration of the loop on line 10. Clearly, \( B(\sigma_1, \alpha), B(\sigma_1, \beta) \in W(\sigma) \) at this point. Choosing \( i = \alpha \) and \( j = i_0 \) on this iteration then since \( \delta(p, \sigma_1) \in B(\sigma_1, \alpha) \) we have

\[
B'_{i_0} = \{ t \in B_{i_0} \mid \delta(t, \sigma_1) \in B(\sigma_1, \alpha) \} \subset B_{i_0}
\]

This holds because \( q \in B_{i_0} \) but \( \delta(q, \sigma_1) \in B(\sigma_1, \beta) \) and \( B(\sigma_1, \alpha) \neq B(\sigma_1, \beta) \) so \( B(\sigma_1, \alpha) \cap B(\sigma_1, \beta) = \emptyset \) and hence \( q \not\in B'_{i_0} \). Therefore \( p \) and \( q \) are split into different blocks on the first iteration so that \( p \in B'_j \) and \( q \in B'_k \).

By symmetry, choosing \( i = \beta \) and \( j = i_0 \) then \( p \) and \( q \) are split on the first loop iteration with \( q \in B'_{i_0} \) and \( p \in B_{i_0} - B'_{i_0} \).

(b) Suppose initially \( \delta(p, \sigma_1), \delta(q, \sigma_1) \in B(\sigma_1, \alpha) \) for some \( \alpha \). Now

\[
\lambda^*(\delta(p, \sigma_1), \sigma_2, \ldots, \sigma_n) \neq \lambda^*(\delta(q, \sigma_1), \sigma_2, \ldots, \sigma_n).
\]

So by the induction hypothesis, eventually \( \delta(p, \sigma_1) \) and \( \delta(q, \sigma_1) \) are split into different blocks, \( \delta(p, \sigma_1) \in B_\alpha \) and \( \delta(p, \sigma_1) \in B_\beta \). At that time one of \( B_\alpha \) or \( B_\beta \) is placed in a waiting set \( W(\sigma) \). Then either on the same iteration of the loop
on line 10 or on the next iteration, we can apply the argument of part (a) again to show that \( p \) and \( q \) are split into different blocks.

Observe that only one split block is loaded into \( W(\sigma) \) on lines 21-24. From the proof of Lemma 6.2 we can see that it does not matter logically which of these two blocks we insert into \( W(\sigma) \). However, by choosing the subset with fewest incoming transitions we can obtain a worst case time complexity of order \( O(|\Sigma|n \log_2 n) \), as we will show.

6.2.5. Corollary. For any states \( p, q \in Q \), if \( p \neq q \) then \( p \) and \( q \) are in different blocks when the algorithm terminates.

Proof. Assume that \( p \neq q \).
(a) Suppose at line 3 that \( n = |Q| \). Then initially, all blocks \( B_i \) are singleton sets and so trivially \( p \) and \( q \) are in different blocks when the algorithm terminates.
(b) Suppose at line 3 that \( n < |Q| \).
(b.i) Suppose that \( p \) and \( q \) are in different blocks initially. Since blocks are never merged then the result holds.
(b.ii) Suppose that \( p \) and \( q \) are in the same block initially. Since \( p \neq q \) then the result follows by Lemma 6.2.

We conclude this section by verifying that our generalisation of Hopcroft’s minimisation algorithm does not actually change its time complexity.

6.2.6. Proposition. If \( A \) has \( n \) states then Algorithm 5 has worst case time complexity \( O(|\Sigma|n \log_2 n) \).

Proof. Creating the initial block partition on line 1 requires at most \( O(n) \) assignments. The block subpartitioning in the loop on line 3 requires at most \( O(kn) \) moves of states. Also the the initialisation of the waiting lists \( W(\sigma) \) in the loop on line 7 requires at most \( O(kn) \) assignments.

Consider one execution of the body of the loop starting on line 10, i.e. lines 13 - 29. Consider any states \( p, q \in Q \) and suppose that \( \delta(p, \sigma) = q \) for some \( \sigma \in \Sigma \). Then the state \( p \) can be: (i) moved into \( B_j' \) (line 15), (ii) removed from \( B_j \) (line 17), or (iii) moved into \( B(\sigma, i) \) or \( B(\sigma, \text{count}) \) (lines 19, 20) if, and only if, a block \( i \) is being removed from \( W(\sigma) \) such that \( q \in B(\sigma, i) \) at that time. (Such a block sub-partition \( B(\sigma, i) \) can be termed a splitter of \( q \).)

Now each time a block \( i \) containing \( q \) is removed from \( W(\sigma) \) its size is less than half of the size when it was originally entered into \( W(\sigma) \), by lines 21-24. So \( i \) can be removed from \( W(\sigma) \) at most \( O(\log_2 n) \) times. Since there are at most \( |\Sigma| \) values of \( \sigma \) and \( n \) values of \( p \), then the total number of state moves between blocks and block sub-partitions is at most \( O(|\Sigma|n \log_2 n) \).

7 Heuristic Estimation of IKL Convergence

When the IKL learning algorithm is applied to the problem of learning based testing of software, the question naturally arises, when should we stop testing? When the system under test (SUT) is sufficiently small, exhaustive testing can
be achieved if we continue until the IKL algorithm converges. But how can we detect convergence?

Traditionally, in automata learning theory, this question is answered by executing an equivalence oracle on the SUT and the hypothesis automaton such as [20]. For a DFA learning algorithm such as L* [2], learning is continued if the equivalence oracle can return a string that is incorrectly learned by the hypothesis DFA, otherwise learning is terminated. However, in the context of black-box testing a glass box equivalence oracle, based on direct comparison of the SUT and the hypothesis automaton, is not acceptable for two reasons:

(1) the principles of black-box testing do not allow us to expose the SUT for glass box equivalence checking, and

(2) even if we ignore (1), in practise there are no glass box equivalence checkers that can compare an arbitrary piece of software (the SUT implementation) with the hypothesis automaton for equivalence.

Of course, a glass box equivalence oracle can be stochastically approximated by a black-box equivalence oracle based on random queries. Random queries are even necessary during LBT when no counterexamples can be found by model checking. However, a purely stochastic solution to equivalence checking is not possible, as we will discuss below. Therefore problems (1) and (2) force us to consider other black-box heuristics for estimating convergence of the IKL algorithm.

---

**Fig. 2.** Graph for True and Estimated Convergence for Elevator
Figure 2 depicts the state space size of successive hypothesis automata $H_i$ ($i = 1, \ldots, 104$) generated by the IKL algorithm while learning and testing a small reactive system against a simple temporal logic specification. In this controlled experiment the SUT was a simplified model of an elevator, with a state space size of 38 states and an input alphabet of 4 symbols. This model is well within the scope of complete learning using IKL, which converges quickly.

It is natural to consider whether any features of a graph such as Figure 2 can be used to estimate the point of convergence. This graph is comparable in its structure for all similar experiments that were conducted. It shows a succession of peaks, each one well above the state space size of the underlying SUT. However at some point these peaks die out and a steady state space size is reached. Each peak and trough seem to indicate a distinct new phase in learning, and therefore they do shed some light on the learning activity. However, they clearly do not indicate convergence, which first appears in hypothesis automaton $H_{55}$. (In controlled experiments we can apply a glass box equivalence checker to accurately determine convergence.)

Although we cannot apply glass box equivalence checking between the SUT and hypothesis automata $H_i$, we can apply it to pairs of successive hypothesis automata $H_i$ and $H_{i-1}$, since the representations of these are known and visible. We can even iterate this test across $n$ successive hypothesis automata $H_1, \ldots, H_{i-n}$ (by conjunction of the outcomes) which we term $n$-equivalence checking. After convergence has been achieved, $n$-equivalence checking will be positive for every value of $n$. This gives a heuristic for black box equivalence checking that is more complex than stochastic equivalence checking, since the queries used to generate successive hypothesis automata are not always random. Many arise from model checking counterexamples. It is difficult to say that queries generated by model checking are randomised, since they are always counterexamples to a specific temporal logic formula, which can strongly bias their structure.

We therefore decided to empirically evaluate the reliability of $n$-equivalence checking as a heuristic indicator of convergence. For this evaluation we considered different SUTs with different state space sizes, different temporal logic formulas, and different values of $n$.

We chose two different SUTs, which were models of a simple cruise controller and a simple elevator. The cruise controller model was an 8-state 5-bit Kripke structure with an input alphabet size of 5. The elevator model was a 38-state 8-bit Kripke structure with an input alphabet size of 4. We considered four different temporal logic test requirements for the cruise controller and six for the elevator. These gave a total of ten convergence experiments for the two SUTs.

Each of these ten experiments was then used to evaluate the $n$-equivalence heuristic for $n = 1, 2, 10, 50$. For $n = 1, 2$ the heuristic completely failed to identify convergence (i.e. the indicator always triggered too early) for all ten experiments. For $n = 10$, just two experiments with the cruise controller (the smaller case study) correctly identified convergence, while eight still failed. Using $n = 50$ all experiments correctly identified convergence. However, note that for increasingly large values of $n$ we tended to overestimate the convergence
point by an increasing margin. Table 1 summarises the relationship between true convergence and estimated convergence for \( n = 50 \).

<table>
<thead>
<tr>
<th>Requirement</th>
<th>True Convergence</th>
<th>Estimated Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( H_{55} )</td>
<td>( H_{104} )</td>
</tr>
<tr>
<td>2</td>
<td>( H_{16} )</td>
<td>( H_{66} )</td>
</tr>
<tr>
<td>3</td>
<td>( H_{44} )</td>
<td>( H_{94} )</td>
</tr>
<tr>
<td>4</td>
<td>( H_{32} )</td>
<td>( H_{82} )</td>
</tr>
<tr>
<td>5</td>
<td>( H_{18} )</td>
<td>( H_{68} )</td>
</tr>
<tr>
<td>6</td>
<td>( H_{25} )</td>
<td>( H_{66} )</td>
</tr>
</tbody>
</table>

**Table 1.** True and Estimated Convergence

These simple experiments suggest that for sufficiently large \( n \), \( n \)-equivalence, can be used as a reliable heuristic indicator for convergence. However, further empirical and theoretical analysis still seems necessary to predict the smallest reliable value of \( n \) which minimises the problem of overestimation.

## 8 Conclusions

We have defined and analysed a learning algorithm IKL for deterministic Kripke structures which is efficient for applications in software testing. This algorithm extends active incremental learning with new features such as lazy learning and projection. We have formally proved the correctness of the IKL algorithm and its main components. We have also empirically evaluated a black box heuristic for detecting convergence of learning, which can be used to terminate testing for small systems under test.

Incremental learning and projection combine to make IKL scalable to larger systems under test. Also, incremental and lazy learning combine to support frequent generation of hypothesis automata with which we can discover SUT errors much faster than random testing by model checking. These claims have been empirically evaluated and supported in [17] and [28]. The IKL algorithm has been implemented in the LBTest tool [18] for learning based testing of reactive systems.

We believe that the efficiency of learning-based testing can be even further improved by more research on model inference. For example, the modular architecture of the IKL algorithm can support experiment with other incremental DFA learning algorithms instead of the ID learning algorithm of Section 4, (e.g. RPNI2 [9]). The impact of the frequency of hypothesis automata generation on testing efficiency could then be further investigated. When hypothesis generation is very frequent the overhead of model checking is high, and this overhead can slow down the entire LBT process. However, if generation is very infrequent,
then little use is made of the model checker to conduct a directed search for SUT errors using queries that can falsify the user requirements. This is also inefficient. (Recall the discussion of Section 1.2.) More generally, we could consider an optimal tuning of the rate of hypothesis automata generation, e.g. based on the estimated density of SUT errors.

The relationship between computational learning and software testing has been a fruitful line of research ever since Weyuker’s thesis [27]. Many fundamental questions remain within the context of learning-based testing. For example, the execution of any automata learning algorithm can always be associated with a prefix tree construction (see e.g. [8]) based on the query set used. How can we influence the choice between breadth-first and depth-first search for SUT errors using this prefix tree? Another important question is whether we can find other techniques to generate active learner queries besides congruence construction? Such techniques should be aimed at reducing the need for random queries, which can be very inefficient in practise.

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


Appendix C

Paper 3 (LBTest: A Learning-based Testing Tool for Reactive Systems)
LBTest: A Learning-based Testing Tool for Reactive Systems

Karl Meinke, Muddassar A. Sindhu

School of Computer Science and Communication, Royal Institute of Technology (KTH), Stockholm 10044, Sweden.

Abstract. We give an introduction to the LBTest tool which implements learning-based testing for reactive systems. It makes use of incremental learning and model checking algorithms to automate: i) test case generation, ii) test execution and iii) test verdict construction. The paper illustrates the tool by means of a pedagogical case study, to enable the user to setup and learn the tool quickly. We provide a usability exercise to support tool evaluation.

1 Introduction

Learning-based testing (LBT) ([1], [2]) is an emerging paradigm for black-box requirements testing that encompasses the three essential steps of: (1) automated test case generation (TCG), (2) test execution, and (3) test verdict (the oracle step). The basic idea of LBT is to automatically generate a large number of high-quality test cases by combining a model checking algorithm with an incremental model inference or learning algorithm. These two algorithms are integrated with the system under test (SUT) in an iterative feedback loop which optimizes test case construction based on previous test outcomes. Test verdicts are constructed by comparing predicted output sequences with observed output sequences.

In this paper we introduce a practical tool, LBTest, for learning-based testing of reactive systems. This tool is intended to serve as a platform for reactive systems testing, that will support the integration of different model checkers and learning algorithms within a single architectural framework. In this way, we hope to support further practical experimentation within the testing community using the LBT paradigm. Our long term aim is to identify good combinations of algorithms which deliver fast, effective and scalable requirements testing. We will describe the architecture of LBTest, and sketch its workflows in terms of the user interface design. Screenshots are included to convey the look and feel of the current release of the tool. The reader is encouraged to download and evaluate the tool. For this purpose we include a simple usability experiment.

The LBT process has the advantage that requirements testing is naturally integrated into the TCG activity. By contrast, construction of random test cases from formal requirements can be a challenging task.

Though no methodologies have yet been published for LBT, the reader should hopefully be able to envisage, from our tool description, how LBTest could be
integrated into various software development lifecycle models. The tool is well suited to agile development processes, since black-box testing is insensitive to structural changes in code. Furthermore testing with LBTest is fully automatic, which means the tool can be run in background mode while other project activities are being pursued.

The structure of this paper is as follows. In Section 2 we give a high-level overview of the LBT method for testing reactive systems. In Section 3, we briefly survey related research and tools from the literature. In Section 4 we provide a tool and workflow description. In Section 5 we describe a usability experiment, which the reader may carry out for themselves. Finally in Section 6, we present some conclusions and describe further work for future releases.

2 Learning-based Testing

To understand the functioning of the LBTest tool, it is helpful to sketch a generic LBT algorithm. At a high-level an LBT algorithm requires three components:

1. A black box system under test (SUT) $S$,
2. a formal user requirement specification $Req$ for $S$ and
3. a learned model $M$ of $S$

Now (1) and (2) are the basic inputs for the algorithm (and our tool), while (3) is internally constructed as the output of a learning algorithm. Learning-based testing is a heuristic iterative method to automatically generate a sequence of test cases. The heuristic idea is to learn a black-box system using tests as queries.

For reactive systems testing, it is conventional to assume that the SUT can be adequately modeled by an automaton or state machine model. This means that requirements modeling, learning and model checking will all focus specifically on automaton models. Therefore, an LBT algorithm for reactive systems testing iterates the following four steps:

(Step 1) Suppose that $n$ test case inputs $i_1,...,i_n$ have been executed on $S$ yielding the system outputs $o_1,...,o_n$. The $n$ input/output pairs $(i_1,o_1),..., (i_n,o_n)$ are synthesized into a learned automaton model $M_n$ of $S$ using an automaton learning algorithm. This step, involving generalization from the given data, gives the possibility to predict previously unseen errors in $S$ during Step 2.

(Step 2) The requirement $Req$ is model checked against the learned model $M_n$ derived in Step 1. This process searches for a counterexample $i_{n+1}$ to the requirement.

(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 $(i_1,o_1),..., (i_{n+1},o_{n+1})$ 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 \(S\).

This algorithm iterates Steps 1...3 until an SUT error is found (Step 4) or execution is terminated. Current criteria for termination in the LBTest tool are either: (i) a bound on the maximum testing time, or (ii) detected convergence in the learning process. When the SUT is sufficiently small to be fully learned, i.e. learning converges, then LBT is a sound and complete method to detect errors. However, for SUTs of industrial size, complete learning may not be feasible. In this case, we have shown in [1] that the use of incremental learning algorithms is essential to achieve scalable testing. The IKL learning algorithm [1] currently used in LBTest has been specifically designed and optimized to deal with this scalability problem.

User requirements on reactive systems that can be used for test case synthesis must be expressed in some kind of formal language. It is widely accepted that temporal logics are well suited to this task, and there exist model checkers for a wide variety of such logics. From the point of view of TCG and counterexample construction, linear temporal logic (LTL) seems particularly well suited. Furthermore, to employ a decision algorithm for the model checking problem (which means that TCG will always terminate in finite time) LBTest currently restricts the user to expressing user requirements in propositional linear temporal logic (PLTL). This language extends propositional logic (i.e. Boolean algebra) with the temporal operators \(G\) (always), \(F\) (eventually), \(X\) (next) and \(U\) (until). Using these connectives many statements about the variant and invariant behaviour of a reactive system can be expressed. In particular, PLTL can express safety properties about an SUT (nothing bad should happen) and liveness properties (something good should happen, e.g. a use case). The capability to test liveness properties of an SUT appears to be a unique feature of LBTest, and has been found to be useful in practise (see [3]). Examples of PLTL specifications will be seen later in Section 5.3.

Some restrictions on the LBTest tool come from the learning algorithms which it uses. An important assumption of most automaton learning algorithms is that the the SUT is deterministic in its behaviour. If the SUT is non deterministic then the appropriate generalisation of these learning algorithms must be used. Furthermore, automaton learning can only be guaranteed to terminate if the input and output alphabets are finite. When a reactive SUT involves infinite data types such as integer or floating point types, (or more generally objects) then these must be finitely approximated using appropriate representative values. Examples of this approximation are given in Sections 4 and 5.

3 Literature and testing tools survey

A tutorial on the basic principles of learning-based testing and their application to different types of SUTs can be found in [2]. The origin of some of these ideas can be found perhaps as far back as [4]. Experimental studies of LBT using different learning and model checking algorithms include [1], [5], [6] and [7]. These experiments have repeatedly shown that LBT can substantially outperform ran-
dom testing as a black-box testing method. Furthermore, the industrial case study of [3], has shown that LBTest is scalable to testing systems of industrial size and complexity.

The adaptive model checking (AMC) approach of ([8]) provides a similar framework to LBTest. However, in AMC, Angluin’s L* learning algorithm is used, which provides very little opportunity for model checking and does not scale to large problems so well, when compared with the IKL incremental learning algorithm of LBTest. Furthermore, AMC uses the Vasilevskii Chow algorithm for conformance testing, which is impractical for large case studies. Other previous works, (e.g. Peled et al. [9] and Raffelt et al. [10]) have also 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. [11] and Chauhan et al. [12]). 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. This related research does not yet seem to have lead to practical testing tools. So LBTest is the first testing tool to be made available that combines automata learning methods with model checker based TCG.

There is of course an extensive literature on using model checkers (without computational learning) to generate test cases for reactive systems (see e.g. the survey [13]). This research has focussed mainly on glass box testing, using structural coverage models. Testing using model checkers is subsumed by the more general and currently popular field of model-based testing (MBT), see e.g. [14]. Practical tools which have emerged from the MBT community include LEIROS Test Generator (LTG/B and LTB/UML) (see [15], [16]), Microsoft’s Spec Explorer (see [17]), Conformiq Designer from Conformiq (see [18]) and University of Waikato’s ModelJUnit (see [16]).

In contrast with model-based testing tools, which perform test case generation using some externally defined model (such as a UML model) LBTest learns (or reverse engineers) its own models for testing purposes. Thus LBTest has the advantage that its models do not have to be manually designed or maintained in parallel with the code development process. Furthermore, the constructed models reflect the actual code at all times. These advantages seem to be consistent with the needs of agile software development processes.

4 Tool Description

4.1 LBTest Architecture

A high level view of learning-based testing was given in Section 2. With this view in mind we can give a more detailed model of the LBTest architecture. This architecture is shown in Fig: 1. It consists of the following five components: i) a PLTL model checker, ii) an instance of an SUT, iii) an oracle, iv) an automaton learning algorithm and v) a random input generator.
We can describe one execution run of the LBTest tool in terms of the above components. It is useful to understand this internal behaviour in order to understand the scope and limitations of LBTest for testing purposes. The learning algorithm will produce the first hypothesis $M_0$ by reading all symbols of the alphabet from the start state and observing the outputs corresponding to these symbols. The hypothesis $M_0$ is the initial hypothesis from which the testing process starts.

This hypothesis is model checked against the LTL requirement formula by the model checker. If the model checker finds a counterexample then this will become the next input $\bar{i}$ to the SUT and will be executed on the SUT to get the observed output $\bar{o}$ and also on the hypothesis automata $M_0$ to get the predicted output $\bar{p}$. Since in this case the input $\bar{i}$ is from a model checker therefore the condition $\bar{i} \in MCQ$ will be true and the oracle will be executed. Both the observed and predicted outputs will be compared by the oracle to give a verdict. The verdict will be a pass if the observed and predicted outputs are not the same. In this case the learning algorithm will continue with hypothesis construction by synthesizing the pair $(\bar{i}, \bar{o})$ to get the next hypothesis $M_1$.

The verdict will be a warning if the observed and predicted outputs are equal but the counterexample $\bar{i}$ has a loop. In this case the learning algorithm will continue with hypothesis construction using the current input/output pair $(\bar{i}, \bar{o})$ to build the next hypothesis $M_1$. The warning verdict arises from testing liveness requirements (such as termination) which can never be demonstrated to fail in finite time.
The verdict will be a *fail* when the predicted and observed outputs are equal and the counterexample does not contain a loop. In this case LBTest execution will stop after giving the *fail* verdict. In the case where the model checker does not find any counterexample to the correctness of PLTL formula the random input generator will provide the next input \( i \) to continue the process of learning and build the next hypothesis \( M_i \).

Iteration of this testing process produces a sequence of hypotheses automata \( M_0, M_1, M_2, \ldots \). If the learning algorithm correctly learns in the limit then the sequence of these automata will eventually converge to the target SUT.

Figure 1 shows a generic architecture for LBTest that can be instantiated by different model checkers and learning algorithms. Currently LBTest supports the IKL learning algorithm and the NuSMV model checker. In the future, we plan to integrate more learning algorithms and other model checkers into the current framework.

### 4.2 LBTest Graphical User Interface

The GUI of the LBTest tool is shown in Fig: 2. It consists of a *menubar* on the top which has a text field to enter PLTL requirements along with other menu items. Just below the *menubar* is a *button pane* which provides quick access buttons to invoke different commands of LBTest which are also available through the *menubar*. Below the *button pane* on the left side of the screen is a *results pane*. Each row in it has five columns and shows information about the hypothesis automata \( M_i \). The column **Hyp No** shows the number \( i \) of the hypothesis automata model \( M_i \) generated. The column **Membership Qs** shows the number of random queries generated for \( M_i \). The column **B/Keeping Qs** shows the number of active learning queries generated by the learning algorithm. The column **Model Check Qs** shows the number of queries (counterexamples) generated by the model checker and the last column **Time(s)** shows the execution time in seconds for each hypothesis generated. Below the *results pane* is an *output pane* which shows different output messages from the LBTest tool during the testing session. To the right of *results table* and below the *button pane* is a *hypothesis pane* which shows a graphical image of the current hypothesis if it has 15 or fewer states.\(^1\)

### 4.3 Setting up of LBTest

To begin practical testing with LBTest several input parameters need to be in place. These include providing LBTest with a specific PLTL requirement formula, specifying the path of the (SUT). This can be a jar file for a java program, or executable files of any other programming language. We must also specify the input and output data types and the path of the model checker on the system where testing is to be done. The location for several files which are

---

\(^1\) hypothesis size of more than 15 states are not shown in the GUI they are only stored on disk.
required/generated by LBTest also needs to be specified before starting testing. These include hypothesis description files in the model checker’s input language which will be read by the model checker to model check against the PLTL requirement formula. The hypothesis images are rendered via the DOT graphics library. A location needs to be provided by the user where these files will be stored for rendering the images in the LBTest GUI. The location of the file to which the results will be written also needs to be supplied by the user. This particular location will be used by the LBTest to write both the text and .xml result files. To terminate a testing session several parameters need to be set. These include specifying the time needed for testing or a convergence measure used for complete learning of smaller systems. The commands necessary to execute the SUT are given to the LBTest tool before starting a testing session. Most of the information about a testing session is saved in a file called “settings.txt”. The location where to store the settings file is given by the user before starting a testing session.

4.4 LBTest Workflow

A brief description of the pre-requisites required to run the LBTest tool was given in Section 4.3. Here we describe in more detail a generic workflow for the

![Fig. 2. The LBTest GUI](image-url)
basic test activity. We then illustrate this workflow with a concrete example based on a cruise controller (cc) application.

The following generic sequence of steps is to be followed by the user for basic testing.

4.4.1 Generic Workflow of Testing

1. **Load SUT:** In the first step the System Under Test (SUT) is loaded into LBTest. An SUT can be any executable (e.g. a jar file or a .exe file) that reads/writes to the console as described in the concrete example in Section 4.4.2.

2. **Define SUT Interface:** This step involves providing the input and output interfaces between the LBTest tool and the SUT.

3. **Input Requirement(s):** In this step, the requirements of the SUT are provided to the LBTest tool in the form of Propositional Linear Temporal Logic (PLTL) formulas which describe the SUT behaviour in terms of the interface defined in Step 2.

4. **Execute:** In this step, the LBTest tool is executed on the SUT of Step 1 to find any violation in behaviour w.r.t. the requirements of Step 3.

5. **Save/Show Results:** In this step the LBTest tool will report and save the results of the testing session.

4.4.2 Workflow Steps with a Concrete Example

To illustrate the workflow steps of the LBTest tool with the help of a concrete example we consider the case of a simple cruise controller (cc). A cc is an embedded safety critical electronic device commonly used in modern vehicles. We consider a simplified cc with a single input data type consisting of five events *
\{brake, dec, gas, acc, button\} and three finite output data types *mode, speed* and *button*.

Table 1 shows the symbolic values of the input data type and its encodings. The first columns shows the symbolic input values of the input data type and their encodings are shown in the second column. The encodings are required for the internal working of the learning algorithm. The encodings are to be given by the user and can be any unicode characters.\(^2\) The symbolic inputs *brake* and *gas* (encoded as ‘a’ and ‘c’) are used to denote deceleration and acceleration of the vehicle respectively while the symbolic input *button* (encoded as ‘e’) is used to turn on or turn off the cc. The inputs *dec* and *acc* (encoded as ‘b’ and ‘d’) denote deceleration and acceleration due to external factors such as the vehicle going uphill or downhill respectively.

Table 2 shows the output data types of the cc and their binary encodings. The cc has a total of five output bits. The first column of the table shows the data types used in the output which are *mode, speed* and *button*. The second and third columns show the start index (inclusive) and the end index (exclusive) of each data type in the output bit vector respectively. The data types *mode, speed* and

\(^2\) Exceptions are unicode characters “\u03bb” and “\u03a6” because these are used in the learning algorithm.
and button have start indices 0, 2 and 4 and end indices 2, 4 and 5 respectively. The symbolic values taken by each data type are shown in the fourth column. These are manual, cruise and disengaged for the mode data type, zero, one and two for the speed data type and on and off for the button data type. The last column of the table shows the binary encodings used for each of the symbolic values of the output data types.

<table>
<thead>
<tr>
<th>Symbolic Input</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>brake</td>
<td>a</td>
</tr>
<tr>
<td>dec</td>
<td>b</td>
</tr>
<tr>
<td>gas</td>
<td>c</td>
</tr>
<tr>
<td>acc</td>
<td>d</td>
</tr>
<tr>
<td>button</td>
<td>e</td>
</tr>
</tbody>
</table>

Table 1. Symbolic Input Encodings

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Start Index</th>
<th>End Index</th>
<th>Symbolic Value</th>
<th>Binary Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>0</td>
<td>2</td>
<td>manual</td>
<td>[false, false]</td>
</tr>
<tr>
<td>mode</td>
<td>0</td>
<td>2</td>
<td>cruise</td>
<td>[false, true]</td>
</tr>
<tr>
<td>mode</td>
<td>0</td>
<td>2</td>
<td>disengaged</td>
<td>[true, false]</td>
</tr>
<tr>
<td>speed</td>
<td>2</td>
<td>4</td>
<td>zero</td>
<td>[false, false]</td>
</tr>
<tr>
<td>speed</td>
<td>2</td>
<td>4</td>
<td>one</td>
<td>[false, true]</td>
</tr>
<tr>
<td>speed</td>
<td>2</td>
<td>4</td>
<td>two</td>
<td>[true, false]</td>
</tr>
<tr>
<td>button</td>
<td>4</td>
<td>5</td>
<td>on</td>
<td>[true]</td>
</tr>
<tr>
<td>button</td>
<td>4</td>
<td>5</td>
<td>off</td>
<td>[false]</td>
</tr>
</tbody>
</table>

Table 2. Output Data Types for the Cruise Controller

A state transition diagram for this cc is shown in Fig 3. To begin testing, the cc model of Fig 3 needs to be made executable. Java code replicating the above cc state transition model can be written and a jar file generated which can act as the SUT for our concrete example. The output of the jar file or any other executable should appear as shown in Fig. 4 on the console. This format implements the “reset assumption” (a form of test setup) used in automaton learning theory (see [19]). A single test execution on the jar file must follow three steps: First, prompting that the cc executable is waiting for input (line 2 in Fig. 4). Second, getting the input string. This can be any input string defined
over the alphabet for the cc (see line 3 in Fig. 4). Third, terminating execution
after showing the output bit label. The string in this case is “brakedecgas” and
it takes the cc to the state with bit label “00010” which can be seen from the
state diagram in Fig. 3. By terminating execution of the jar file we ensure that
the cc is returned to its initial state on the next test execution.

Now with an abstract input and output interface available (Tables 1 and 2)
along with the executable code we are ready to carry out the generic steps given
in Section 4.4.1 on our cc example.

1. **Load SUT:** This can be done either through the menu bar by choosing SUT
   -> Get SUT or through the button pane by clicking the SUT button. A file
   chooser dialog box will appear where the user needs to select the directory
   where the jar/executable file of the SUT is located. In the specific case of
testing a jar file a console execution command usually requires a sequence of strings to specify execution options. These can be provided by choosing either Settings -> Run Options or by clicking the Options button on the button pane. This will open the dialog box shown in Fig. 5. In this case the command options include “java”, “-jar” and “cruiseCorrect.jar” as shown in Fig.5. The order of options is numbered 1 to 5 in this figure.

2. Define Interface: This has two parts: First the input alphabet of Table 1 is provided and secondly the output data types of Table 2 are defined.

   (a) Input Alphabet: This can be provided to the LBTest tool either through the menu bar by choosing SUT -> Alphabet or through the button pane by clicking the Alphabet button. An Input Dialog similar to the one in Fig 6. will open. Here the information in Table 1 can be inserted row by row. Adding and removing rows from the table is straight forward. The user can add a new row by providing a new value of symbolic input alphabet and its encoding in the designated fields and pressing the green
(+) button or just clicking the return key on the keyboard. A row can be removed by selecting it first in the table and then pressing the red (-) button. When the user is happy with the data in the table it can be saved by clicking the Save button.

(b) Output Data: To provide the output data type information the user can again use either the menu bar SUT -> Output or through the button pane by clicking the Output button. This will open an Output Dialog as shown in Fig. 7. Here the information in Table 2 can be inserted row by row. There are text fields/ spinners for the first four columns where the user can give the desired values. Afterwards to specify the Boolean values of bits in a row the user has to click Set Bits and an appropriate number of checkboxes will appear. Where the user can set the values and then click Add Row button to insert this row into the model. Once the user is happy with the settings the model can be saved by clicking Save Model button. The latest saved model can be reloaded for a future testing session.

3. Input Requirement: At this point the user can input a user requirement in the LTL text field at the top of LBTest GUI. For example, one safety property of the cruise controller is that the vehicle is disengaged from the cruise mode when the gas pedal is pressed. This property can be specified in terms of the interface definitions of Table 1 and Table 2 as:

\[ G\left( mode = cruise \land letter = gas \rightarrow X\left( mode = disengaged \right) \right) \]

4. Execute: After loading all the pre-requisites in the previous three steps the user can now start testing by pressing the Start Testing button near the bottom of LBTest interface. The results will begin to show up in the Results Table after a while.

5. Save/Show Results: Once a testing session is finished the contents of the Results Table can be saved to a file by pressing the Save Results button under the Results Table. This will create a file results.txt in the chosen location.
4.4.3 Error Discovery with LBTest The preceding section described the use case to setup LBTest for black box testing of requirements. In this section we describe a use case where an error exists in the SUT that violates the PLTL property given in Step 3. For this, we can inject an error into the cc as a mutation which ensures that the cc does not go to the disengaged mode when the gas pedal is pressed from the cruise mode. The output of the mutated jar file is shown in Fig. 8. In response to the input string “deccgasbuttongas” the mutated jar file outputs a bit label “01011” instead of “10101” as defined in the state transition diagram of Fig. 3. When LBTest is run on this mutated SUT it correctly locates this bug after 467 seconds as shown in Fig. 9 with the counterexample “gasbuttongasgasdecaccdecdec”. With this counterexample as input the mutated SUT does not go out of the cruise mode from the third symbol gas onwards see state 10 of the last hypothesis in the hypothesis pane (in Fig. 9) and remains in the loop on the same state in the cruise mode. While in the defined model of Fig. 3 it disengages from the cruise mode on receiving symbolic input gas as specified by the PLTL property.

The verdict of a test case is shown by the colour of the Verdict Box at the bottom of the LBTest GUI (see Fig 9). A green box shows a passed test case, a yellow box indicates a warning (i.e a loop in the counterexample) as with our current example and a red box indicates a failed test case. The box will remain grey if no counterexample to the correctness of PLTL property is found at all during the whole testing session. like Fig. 2 shows a snapshot of a finished testing session of one hour with no violation of behaviour of the specified PLTL property discovered during this time.

5 A Usability Exercise

In the preceding sections we have illustrated the working of LBTest with the help of a simplified case study. In this section we provide a usability exercise for users, which they can experiment with to evaluate LBTest.
5.1 Aims

This usability exercise aims to enable the user to think about the following factors that can affect the behaviour of the tool in finding errors. These include:

– the structure of the requirement formula (e.g. How many variables and logical connectives are there in the formula and how are they nested?)
– the actual location of the error in the system (i.e. How close the error is located to the initial state?)
– the number, nature and timing of random queries (Which paths of the system these enable the learning algorithm to explore?)

The location of the error inside the system can be a significant factor in error discovery time. The tool may find errors that lie closer to the initial state more quickly than those which lie farther away.

Similarly, the number, nature and timing of random queries is also very important because eventually they take the learning algorithm either towards the path to the error or along an entirely different path.

5.2 Usability Exercise Activity

The user should carry out the following sequence of activities to perform two usability exercises.
5.2.1 Setup SUT To setup a testing experiment the user will need an SUT (jar file / executable code). In setting up the SUT the user should think about the data interfaces for input and output and the reset option. By the reset option we mean returning the SUT to the initial state after each test.

5.2.2 Define Input Interface In this step the user should identify the input data types and assign them some symbolic values as described in Section 2(a).

5.2.3 Define Output Interface In this step the user should identify the total number of bits in the output, output data types, bits required per data type, their symbolic values and binary encodings for each symbolic value of the data types. Once the analysis is complete the user can then proceed according to the description of Section 2(b).

5.2.4 Formalize Requirements During this activity the user is required to write valid PLTL formulas corresponding to each of the informal requirements provided Section 5.4.2 in terms of the interface definitions.

5.2.5 Error Discovery The user can analyze the error discovery feature of LBTest by injecting errors into different parts of the SUT. While injecting errors and analyzing their discovery through LBTest the user should consider the points of influence mentioned in Section 5.1.

5.3 Exercise 1: CC Formal Requirements

Just to give the user a taste of PLTL requirements and how they are derived from informal requirements in terms of interface definitions of the SUT in LBTest, we can consider two more informal requirements on the cc discussed in Section 4. We give the corresponding PLTL requirements for each.

1. The vehicle maintains cruise speed while going uphill in the cruise mode.

   \[ G(\text{mode} = \text{cruise} \& \text{speed} = \text{one} \& \text{letter} = \text{dec} \rightarrow X(\text{speed} = \text{one})) \]

2. The vehicle disengages when brake is pressed in the cruise mode.

   \[ G(\text{mode} = \text{cruise} \& \text{letter} = \text{break} \rightarrow X(\text{mode} = \text{disengaged})) \]

5.4 Exercise 2: Elevator Model and Requirements

This exercise concerns an n-floor elevator system.
5.4.1 Problem Description An \( n \)-floor elevator system consists of one physical elevator vehicle and \( n \) physical floors. The elevator vehicle has \( n \) command buttons and pressing the \( i \)-th command button issues a request for the elevator to visit the \( i \)-th floor. The elevator vehicle also has open and close buttons which activate the doors, but only when the elevator is stationery and on a physical floor. Each physical floor has a request button. Pressing a request button on the \( i \)-th floor issues a request for the elevator vehicle to visit the \( i \)-th floor.

Any button can be pressed at any time.

5.4.2 Informal Requirements Keeping in view the context of the preceding paragraph following set of informal requirements should be formalized and tested on an SUT implementation:

1. When the \( i \)-th request button is pressed then the vehicle should arrive at the \( i \)-th physical floor.
2. When the \( i \)-th command button is pressed then the vehicle should arrive at the \( i \)-th physical floor.
3. The vehicle door is always closed when the vehicle is moving.
4. The vehicle is never stopped when not at a floor.
5. When the vehicle arrives on a floor then the door automatically opens.
6. If the vehicle is stationary at a floor with the door closed and that floor is requested, then the vehicle should stay at that floor but should open the door.
7. If the vehicle is stationary at a floor with the door closed and no requests or commands are made or pending, then the vehicle should stay at that floor with the door closed.

6 Conclusions and Future Work

We have given an introduction to the basic theoretical principles used in the development of a new tool LBTest for learning-based requirements testing of reactive systems. Its functionality has been explained with the help of a pedagogical case study. A usability exercise has been provided for the users to evaluate the tool and their own learning experience. Future research will include compiling a collection of industrial case studies and experiences. We also plan to integrate more learning and model checking algorithms into the platform.

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 and ARTEMIS project 269335 MBAT.

References


Appendix D

Paper 4 (Case Studies in Learning-based Testing)
Case Studies in Learning-based Testing

Lei Feng\textsuperscript{1}, Karl Meinke\textsuperscript{2}, Fei Niu\textsuperscript{2},
Muddassar A. Sindhu\textsuperscript{2}, and Peter Y. H. Wong\textsuperscript{3}

\textsuperscript{1}Machine Design Department, Royal Institute of Technology, Stockholm 10044, Sweden. feng@kth.se,
\textsuperscript{2}School of Computer Science and Communication, Royal Institute of Technology, Stockholm 10044, Sweden. \{karlm, niu, sindhu\}@csc.kth.se
\textsuperscript{3}SDL Fredhopper, Amsterdam, The Netherlands. peter.wong@fredhopper.com

Abstract. We present case studies which show how the paradigm of learning-based testing (LBT) can be successfully applied to black-box requirements testing of reactive systems. For this we apply a new testing tool \textit{LBTest}, which combines algorithms for incremental black-box learning of Kripke structures with model checking technology. We show how test requirements can be modeled in propositional linear temporal logic extended by finite abstract data types. We provide benchmark performance results for LBTest applied to two industrial case studies. Finally we present a first coverage study for the tool.

1 Introduction

Learning-based testing (LBT) is an emerging paradigm for \textit{black-box requirements testing} that automates the three basic steps of testing: (1) automated test case generation (ATCG), (2) test execution, and (3) test verdict (the oracle step). It has been successfully applied to testing \textit{procedural systems} in [12]. The first application of LBT to testing reactive systems was given in [17] and [14]. A tutorial on the LBT paradigm, which compares it with related approaches is [15].

The basic idea of LBT is to automatically generate a large number of high-quality test cases by combining a model checking algorithm with an \textit{incremental model inference} or \textit{learning algorithm}. These two algorithms are integrated with the system under test (SUT) in an iterative feedback loop. On each iteration of this loop, a new test case can be generated either by: (i) model checking a learned model $M_i$ of the system under test (SUT) against a formal user requirement $req$ and choosing any counterexample to correctness, (ii) using the learning algorithm to generate a membership query, or (iii) random generation. Whichever method is chosen, the new test case $t_i$ is then executed on the SUT, and the outcome is judged as a pass, fail or warning. This is done by comparing a predicted output $p_i$ (obtained from $M_i$) with the observed output $o_i$ (from the SUT). The new input/output pair $(t_i, o_i)$ is also used to update the current model $M_i$ to a refined model $M_{i+1}$, which ensures that the iteration can proceed again. If the learning algorithm can be guaranteed to correctly learn in the limit, given
enough information about the SUT, then LBT is a sound and complete method of testing. In practice, real-world systems are often too large for complete learning to be accomplished within a feasible timescale. By using incremental learning algorithms, that can focus on learning just that part of the SUT which is relevant to the requirement req, LBT becomes much more effective.

While algorithms for LBT have been analyzed and benchmarked on small scale academic case studies (see [17], [14]), there has so far been a lack of evaluation on real-world case studies. The work presented here therefore, has three aims:

1. to describe the requirements language of LBTest in detail with case studies,
2. to show that the positive testing results previously obtained for small academic case studies do indeed scale up to larger industrial case studies, and
3. to provide a first coverage study for LBTest.

The organization of this paper is as follows: In Section 3 we give an introduction to requirements testing with the LBTest tool by providing an introduction to its requirements language and input/output interfaces. In Section 4 we describe two industrial case studies with the LBTest tool. In Section 5 we discuss some coverage results achieved with LBTest and finally in Section 6 we give some conclusions and future directions of research.

2 Literature and tool survey

A tutorial on the basic principles of LBT and their application to different types of SUTs can be found in [15]. The origin of some of these ideas can be traced perhaps as far back as [22]. Experimental studies of LBT using different learning and model checking algorithms include [17], [12], [13] and [14]. These experiments have repeatedly shown that LBT can substantially outperform random testing as a black-box testing method.

Several previous works, (for example Peled et al. [18], Groce et al. [10] and Raffelt et al. [19]) have also 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. [7] and Chauhan 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. This related research does not yet seem to have lead to practical testing tools. So LBTest is the first experimental testing tool to be made available that combines automata learning methods with model checker based TCG.

There is of course an extensive literature on using model checkers (without computational learning) to generate test cases for reactive systems (see e.g. the survey [9]). This research has focussed mainly on glass box testing, using structural coverage models. Testing using model checkers is subsumed by the more
general and currently popular field of model-based testing (MBT), see e.g. [8]. Practical tools which have emerged from the MBT community include Conformiq Designer from Conformiq (see [1]), University of Waikato’s ModelJUnit (see [20]), LEIROS Test Generator (LTG/B and LTB/UML) (see [2], [20]) and Microsoft’s Spec Explorer (see [21]).

In contrast with model-based testing tools, which perform test case generation using some externally defined model (such as a UML model) LBTest learns (or reverse engineers) its own models for testing purposes. Thus LBTest has the advantage that its models do not have to be manually designed or maintained in parallel with the code development process.

3 Requirements Testing with LBTest

A platform for learning-based testing known as LBTest (see [16]) has been developed within the EU FP7 HATS project (ref). This platform supports black-box requirements testing of fairly general types of reactive systems. The only general requirement for applying LBTest is that it must be possible to model a particular SUT by a finite state machine abstraction. For research purposes, LBTest supports the integration of different model inference algorithms with different model checkers to evaluate new learning-based testing algorithms.

The main inputs to LBTest are a black box SUT and a set of formal user requirements to be tested. The tool is capable of generating executing and judging tens of thousands of tests cases per hour, with the main limitation on throughput being the average execution time of an individual test case on the SUT.

For user requirements modeling, the formal language currently supported in LBTest is propositional linear temporal logic (PLTL) extended by finite data types. In particular, PLTL formulas can express both: (i) safety properties which are invariants that may not be violated, and (ii) liveness properties, including use cases, which specify intended dynamic behaviors. A significant new contribution of LBTest is its support for liveness testing. Our case studies in Section 3 will display typical examples of both safety and liveness properties, expressed in PLTL.

Note that currently in LBTest, only one model checker interface is supported, which is an interface to the NuSMV model checker (see e.g. [6]). Further interfaces are planned in the future. The learning algorithm currently available in LBTest is the IKL learning algorithm described in [17], which is an algorithm for learning deterministic Kripke structures. New learning algorithms are currently in development for future evaluation.

3.1 PLTL as a Requirements Modeling Language

In the context of reactive systems analysis, temporal logics have been widely used to model user requirements. Indeed, semi-formal user requirements, languages such as UML sequence diagrams, can be given a precise semantics in terms of temporal logic.
From a testing perspective, linear time temporal logic (LTL) with its emphasis on the properties of paths or execution sequences, is a natural choice from among the diversity of known temporal logics. Since the design philosophy of LBTest is to generate, execute and judge as many test cases as possible within a given time frame, this places stringent requirements on the efficiency of model checking LTL formulas. Therefore, only model checking of propositional linear temporal logic (PLTL) formulas has been considered so far. Basic PLTL supports only the Boolean data type, and is more oriented towards testing control properties, than functional properties. Since this capability was felt to be too restrictive in many practical case studies, we have extended PLTL with the addition of user defined (symbolic) finite data types. Such finite data types are intended to bridge the gap between the fixed length bit vector encodings possible in Boolean logic and general infinite data types such as integers, strings and floating point numbers. This gap is also bridged by providing formal logical support for partition testing, to deal specifically with infinite data types.

To use the LBTest tool correctly it is important to understand the precise syntax of PLTL which is supported. Furthermore, to understand the case studies of Section 4, and to conduct new case studies, it is important to have at least an informal understanding of PLTL semantics. Therefore we shall define these two aspects of the LBTest language interface in this section.

Before formally defining the extended PLTL syntax supported by LBTest, we need to precisely define our data type model. This is based on the well known algebraic model of abstract data types, involving many-sorted signatures and algebras (see e.g. [11]). To ensure that the model checking problem for this extension of PLTL remains decidable, we support only finite abstract data types. (Thus we do not support constructors, data operations or selection functions, however see [14]). In Section 3, we consider how such finite data type models can be combined with externally defined partitioning relations to abstract infinite data types into finite ones. The second case study of Section 4 (brake-by-wire) requires this capability to handle the floating point data type. Thus it provides a good example of using partitioning and data abstraction to approximate infinite data types by finite ones.

3.1.1 Definition (i) A finite data type signature $\Sigma$ consists of a finite set $S$ of sorts or types, and for each sort $s \in S$, a finite set $\Sigma_s$ of constant symbols all of the same type $s$. (ii) If $\Sigma$ is a finite $S$-sorted data type signature then a concrete $\Sigma$ data type $A$ consists of: (a) a family of finite sets $A_s$ for each sort $s \in S$, and (b) for each sort $s \in S$ and each symbolic constant $c \in \Sigma_s$ a concrete value $c_A \in A_s$.

In practice, a type $s \in S$ may either refer to an event type (e.g. mouse action) or a data type (e.g. int, string), however the generic term data type is preferred. A symbolic constant symbol $c \in \Sigma_s$ is a symbolic name for an important (from a testing perspective) concrete value from a concrete data domain $A_s$. These named symbolic constants may exhaust the domain of possible values (if this
domain is finite and not excessively large), or they may simply sample the domain at strategic points. Examples of both situations can be seen in Section 3.

Note that the important principle of data abstraction encapsulated in this separation of symbolic data names from concrete data values is an important mechanism in LBTest. Besides adding clarity and abstraction on the level of formal requirements modeling, abstract data types also clarify the role and definition of the data communication protocol that must be used between LBTest and the SUT during each testing session. This protocol is supported by LBTest in its role as a client (client side communication) automatically through user defined data type encoding tables. (Pedagogical examples of such tables can be seen in the next Section 2.3, and also Section 3.) For the SUT in its role as a server (server side communication) a simple wrapper program must be written by the tester to support data conversion on the server side. In effect, this also supplies a data type encoding table to the SUT. Both data type encodings (client and server) are simply different concrete implementations of the same abstract data type. This formal approach is necessary when we consider that while LBTest is implemented in Java, the SUT can be written in an arbitrary programming language with its own concrete data types.

We now define the syntax of PLTL extended by a finite data type signature $\Sigma$.

3.1.2 Syntax of PLTL($\Sigma$) Let $S$ be a finite set of sorts containing a distinguished input type $in \in S$, and let $\Sigma$ be a finite data type signature. The syntax of the language PLTL($\Sigma$) of extended propositional linear temporal logic over $\Sigma$ has the following Backus Naur Form (BNF) definition:

$$
\phi ::= \bot \mid \top \mid s = c \mid s \neq c \mid (\neg \phi) \mid (\phi_1 \land \phi_2) \mid (\phi_1 \lor \phi_2) \mid (\phi_1 \rightarrow \phi_2) \mid (X \phi) \mid (F \phi) \mid (G \phi) \mid (\phi_1 U \phi_2) \mid (\phi_1 W \phi_2) \mid (\phi_1 R \phi_2)
$$

where $s \in S$ and $c \in \Sigma_s$. (Thus the language has a simple but strict typing system.)

The atomic formulas of PLTL($\Sigma$) are equations and inequations over the data type signature $\Sigma$ for defining input and output constraints. Note that only a single variable of each type is allowed. This variable is synonymous with its associated type, and is the unique variable for writing or reading data of that type. The distinguished input type $in \in S$ also denotes the single input variable for data of type $in$. Every other type $s \in S$ denotes an output variable for reading values of type $s$.

The constants and operations $\bot$, $\top$, $\neg$, $\land$, $\lor$ and $\rightarrow$ are the usual Boolean constants and connectives which have their conventional meaning. The operators $X$, $F$, $G$, $U$, $W$ and $R$ are the temporal connectives. Here $X\phi$ means that $\phi$ is true in the next state, $F\phi$ means that $\phi$ is true sometime in the future, $G\phi$ means that $\phi$ is always true in the future (including the present) and $U$ is the binary operator which means that $\phi_1$ will remain true until (strong until) a point in the
future when $\phi_2$ becomes true. The two operators $W$ and $R$ stand for *weak until* and *release* respectively.

### 3.2 Finite Data Type Modeling and Interface Definitions

For the purpose of explaining finite data type modeling in *LBTest* we will consider a small pedagogical example of a simplified *cruise controller* (CC). A CC is an embedded safety critical component commonly used in modern vehicles. Our simplified CC model involves four data types, $S = \{in, mode, speed, button\}$.

The input data type consists of five event names which are given by, $\Sigma_{in} = \{brake, dec, gas, acc, button\}$. The symbolic inputs brake and gas are used to denote deceleration and acceleration events issued by the vehicle driver, while the symbolic input button is used to turn on or turn off the CC. The input events dec and acc denote deceleration and acceleration events from the external environment due to physical factors such as moving uphill or downhill respectively.

Table 1 shows the symbolic values of the input data type (column one) and their encodings (column two).

<table>
<thead>
<tr>
<th>Symbolic Value</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td></td>
</tr>
<tr>
<td>speed</td>
<td></td>
</tr>
<tr>
<td>button</td>
<td></td>
</tr>
</tbody>
</table>

These symbolic values and encodings must be entered into LBTest (for example as a setup file) by the user. The encodings can be any unicode characters.

Table 2 shows the symbolic values of all output data types for the CC. This table identifies three output types *mode*, *speed* and *button* in column one. These types require 2, 2 and 1 bit for encoding values respectively. So a total of 5 bits are required to encode an output vector from the CC, consisting of one value from each type.

The second and third columns of Table 2 show the start index (inclusive) and the end index (exclusive) of each bit encoding needed for the communication protocol. The data types *mode*, *speed* and *button* have start indices 0, 2 and 4 and end indices 2, 4 and 5 respectively.

The symbolic values taken by each data type are shown in the fourth column of Table 2. These are *manual*, *cruise* and *disengaged* for the *mode* data type and *on* and *off* for the *button* data type. The last column of Table 2 shows the binary encoding used for each symbolic value of each output data type.

Since vehicle velocity would normally be represented by an infinite data type such as the floating point type, the *speed* data type must represent a discretisation of this infinite data type. Such a discretisation must partition the infinite set of values into a finite set of equivalence classes, and precisely define membership of each partition class. For this we define a discretisation formula for each partition class. Such formulas must then be implemented within the SUT wrapper (server side) to allocate a symbolic output value to each observed SUT output value. In this way LBTest also provides support for partition testing. Given the state of the art in model checking technology partitioning seems to be essential for infinite data types. (Though see our remarks in Conclusions Section 6)

So the *speed* data type is a discretisation of an infinite data type (possibly modeled by fixed, floating point or even integer values within the SUT). In this pedagogical example, we provide a very coarse discretisation consisting of just three partition classes, symbolically named *slow*, *cruise* and *fast*. Intuitively
cruise is an acceptable range of cruising speeds, while the other classes represent speeds too slow or too fast for cruising. Discretisation formulas for these symbolic values might for example be $0.0 \leq speed < 60.0$ (slow), $60.0 \leq speed \leq 110.0$ (cruise) and $110.0 < speed$ (fast). Such formulas are easily implemented within the SUT wrapper to return symbolic values from the SUT at test execution time.

<table>
<thead>
<tr>
<th>Constant Symbol</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>brake</td>
<td>a</td>
</tr>
<tr>
<td>dec</td>
<td>b</td>
</tr>
<tr>
<td>gas</td>
<td>c</td>
</tr>
<tr>
<td>acc</td>
<td>d</td>
</tr>
<tr>
<td>button</td>
<td>e</td>
</tr>
</tbody>
</table>

Table 1. Input Data Type for CC

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Start index</th>
<th>End index</th>
<th>Symbolic Value</th>
<th>Binary Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>0</td>
<td>2</td>
<td>manual</td>
<td>[f, f]</td>
</tr>
<tr>
<td>mode</td>
<td>0</td>
<td>2</td>
<td>cruise</td>
<td>[f, t]</td>
</tr>
<tr>
<td>mode</td>
<td>0</td>
<td>2</td>
<td>disengaged</td>
<td>[t, f]</td>
</tr>
<tr>
<td>speed</td>
<td>2</td>
<td>4</td>
<td>slow</td>
<td>[f, f]</td>
</tr>
<tr>
<td>speed</td>
<td>2</td>
<td>4</td>
<td>cruise</td>
<td>[f, t]</td>
</tr>
<tr>
<td>speed</td>
<td>2</td>
<td>4</td>
<td>fast</td>
<td>[t, f]</td>
</tr>
<tr>
<td>button</td>
<td>4</td>
<td>5</td>
<td>on</td>
<td>[f]</td>
</tr>
<tr>
<td>button</td>
<td>4</td>
<td>5</td>
<td>off</td>
<td>[f]</td>
</tr>
</tbody>
</table>

Table 2. Output Data Types for the Cruise Controller

A state transition diagram for this CC is shown in Fig 1.

4 Case Studies in Learning-Based Testing

In this section we present two industrial case studies which were tested with LBTest. These are: i) an access server from Fredhopper (FAS), and ii) a break-by-wire (BBW) system from Volvo Technology. Both these case studies represent mature applications from quite different industrial domains yet the tool was able to find errors in both of them, which is a promising achievement. The following features are important in these case studies:
— The FAS is an e-Commerce application while the BBW is an embedded application from the automobile industry.
— The FAS has been developed and evolved over 12 years and its various modules have been tested with automated and manual techniques. The BBW is relatively newer and is not widely adopted yet.
— The FAS does not have very strict timing constraints. The BBW has very strict timing constraints to ensure the safety of the vehicle.
— The FAS is very complex in terms of the input alphabet and lines of code. The BBW is smaller on both these counts.
— The FAS only involves events and finite data types while the BBW involves events and infinite data types.

The last point mentioned above really sets these case studies apart because the arrival of data along with events makes the BBW case study very challenging to test and requires a different approach to set it up for testing with LBTest. It requires discretizing the infinite data type by using a discretization formula. We gave an introduction to this approach in Section 3.2 and we will explain it further in Section 4. For each case study, we begin with an informal description, then we describe the results obtained, and finally we analyze and explain the results and errors found in both these case studies.

4.1 Case Study 1: Access Server

The Fredhopper Access Server (FAS) is a distributed, concurrent OO system developed by Fredhopper that provides search and merchandising services to e-
Commerce companies, including structured search capabilities within the client’s data. Fig. 1(a) shows the deployment architecture used to deploy an FAS to a customer. An FAS consists of a set of live environments and a single staging environment. A live environment processes queries from client web applications via web services. A staging environment is responsible for receiving data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments according to the Replication Protocol. The Replication Protocol is implemented by the Replication System which consists of a SyncServer at the staging environment and one SyncClient for each live environment. The SyncServer determines the schedule of replication jobs, as well as their contents, while SyncClient receives data and configuration updates according to the schedule. Fig. 1(b) shows the interactions in the Replication System. Informally, the Replication Protocol is as follows: the SyncServer begins by listening for connections from SyncClients. A SyncClient creates and schedules a ClientJob object with job type Boot that connects immediately to the SyncServer. The SyncServer then creates a ConnectionThread to communicate with the SyncClient’s ClientJob. The ClientJob asks the ConnectionThread for replication schedules, notifies the SyncClient about the schedules, receives a sequence of file updates according to the schedule from the ConnectionThread and terminates.

The existing QA practise at Fredhopper is to run a daily QA process. The core component (~160,000 LoC) of FAS, including the Replication System has 2500+ unit tests (more with other parts of FAS). There is also a continuous build system that runs the unit tests and a set of 200+ black box test cases automated using WebDriver (see [3] ) for every code change / 3rd library change to FAS. Moreover, for every bug fix or feature addition, specific manual test cases are run by a QA team and for every release, a subset of standard manual test cases (900+) is executed by the QA team.

The LBTest tool was applied to the problem of black-box testing a Java model of the FAS. This model consisted of about 6400 lines of Java code organized into 44 classes and 2 interfaces. Specifically, we were interested to test the interaction...
between SyncClient and ClientJob by learning a 10-bit Kripke structure over the following input data type 

\[ \Sigma_{in} = \{\text{setAcceptor, schedule, searchJob, businessJob, dataJob, connectThread, noConnectionThread}\} \]

This small input set suffices to model the interaction between the FAS and its live environment. Table 4 shows the binary encoding of the output data types using 10 bits. Eleven informal user requirements were then formalized in PLTL as follows.

**Requirement 1:** If the SyncClient is at state \( \text{Start} \) and receives an acceptor, the client will proceed to state \( \text{WaitToBoot} \) and execute a boot job.

\[ G(\text{state} = \text{Start} \land \text{in} = \text{setAcceptor} \rightarrow X(\text{state} = \text{WaitToBoot} \land \text{jobtype} = \text{Boot})) \]

**Requirement 2:** If the SyncClient’s state is either \( \text{WaitToBoot} \) or \( \text{Booting} \) then it must have a boot job (\( \text{Jobtype} = \text{Boot} \)), and if it has a boot job, its state can only be one of \( \text{WaitToBoot, Booting, WaitToReplicate or End} \).\(^4\)

\[ G(\text{state} \in \{\text{WaitToBoot, Booting}\} \rightarrow \text{jobtype} = \text{Boot} \rightarrow (\text{state} \in \{\text{WaitToBoot, Booting, WaitToReplicate, End}\})) \]

**Requirement 3:** If the SyncClient is executing a Boot job (\( \text{Jobtype} = \text{Boot} \)) and is in state \( \text{WaitToBoot} \) and receives a connection to a connection thread, it will proceed to state \( \text{Booting} \).

\[ G(\text{jobtype} = \text{Boot} \land \text{state} = \text{WaitToBoot} \land \text{in} = \text{connectThread} \rightarrow X(\text{jobtype} = \text{Boot} \land \text{state} = \text{Booting})) \]

\(^4\) The membership relation \( \in \) used in requirement 2 and elsewhere does not belong to PLTL(\( \Sigma \)) but is a macro notation that is replaced automatically.
<table>
<thead>
<tr>
<th>Data Type</th>
<th>Start Index</th>
<th>End Index</th>
<th>Symbolic Output</th>
<th>Binary Encoding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schedules</td>
<td>0</td>
<td>3</td>
<td>φ</td>
<td>[f, f, f]</td>
<td>Specifies the replication schedules to which the SyncClient should commit at any time.</td>
</tr>
<tr>
<td>schedules</td>
<td>0</td>
<td>3</td>
<td>{search}</td>
<td>[f, f, f]</td>
<td></td>
</tr>
<tr>
<td>schedules</td>
<td>0</td>
<td>3</td>
<td>{business}</td>
<td>[f, t, f]</td>
<td></td>
</tr>
<tr>
<td>schedules</td>
<td>0</td>
<td>3</td>
<td>{business, search}</td>
<td>[f, t, f]</td>
<td></td>
</tr>
<tr>
<td>schedules</td>
<td>0</td>
<td>3</td>
<td>{data}</td>
<td>[t, f, f]</td>
<td></td>
</tr>
<tr>
<td>schedules</td>
<td>0</td>
<td>3</td>
<td>{data, search}</td>
<td>[t, f, t]</td>
<td></td>
</tr>
<tr>
<td>schedules</td>
<td>0</td>
<td>3</td>
<td>{data, business}</td>
<td>[t, t, f]</td>
<td></td>
</tr>
<tr>
<td>schedules</td>
<td>0</td>
<td>3</td>
<td>{data, business, search}</td>
<td>[t, t, t]</td>
<td></td>
</tr>
<tr>
<td>state</td>
<td>3</td>
<td>6</td>
<td>Start</td>
<td>[f, f, f]</td>
<td>Specifies the state which the SyncClient is in as specified by the SyncClient State Machine.</td>
</tr>
<tr>
<td>state</td>
<td>3</td>
<td>6</td>
<td>WaitToBoot</td>
<td>[f, f, f]</td>
<td></td>
</tr>
<tr>
<td>state</td>
<td>3</td>
<td>6</td>
<td>Boot</td>
<td>[f, t, f]</td>
<td></td>
</tr>
<tr>
<td>state</td>
<td>3</td>
<td>6</td>
<td>WaitToReplicate</td>
<td>[f, t, f]</td>
<td></td>
</tr>
<tr>
<td>state</td>
<td>3</td>
<td>6</td>
<td>WorkOnReplicate</td>
<td>[t, f, f]</td>
<td></td>
</tr>
<tr>
<td>state</td>
<td>3</td>
<td>6</td>
<td>End</td>
<td>[t, f, t]</td>
<td></td>
</tr>
<tr>
<td>jobtype</td>
<td>6</td>
<td>9</td>
<td>nojob</td>
<td>[f, f, f]</td>
<td>Specifies the type of client job scheduled by the SyncClient according to the replication schedules received.</td>
</tr>
<tr>
<td>jobtype</td>
<td>6</td>
<td>9</td>
<td>Boot</td>
<td>[f, f, f]</td>
<td></td>
</tr>
<tr>
<td>jobtype</td>
<td>6</td>
<td>9</td>
<td>SR</td>
<td>[f, t, f]</td>
<td></td>
</tr>
<tr>
<td>jobtype</td>
<td>6</td>
<td>9</td>
<td>BR</td>
<td>[f, t, f]</td>
<td></td>
</tr>
<tr>
<td>jobtype</td>
<td>6</td>
<td>9</td>
<td>DR</td>
<td>[t, f, f]</td>
<td></td>
</tr>
<tr>
<td>files</td>
<td>9</td>
<td>10</td>
<td>readonly</td>
<td>[f]</td>
<td>Specifies whether the file system be written to by the SyncClient.</td>
</tr>
<tr>
<td>files</td>
<td>9</td>
<td>10</td>
<td>writable</td>
<td>[f]</td>
<td></td>
</tr>
</tbody>
</table>

Table 4. SyncClient Output Data Types
Requirement 4: If the SyncClient is executing a Boot job (Jobtype = Boot) and is in state Booting and receives schedules (schedule), it will proceed to state WaitToReplicate and it will queue all schedules (schedules = {data, business, search}).

\[ G(\text{jobtype} = \text{Boot} \land \text{state} = \text{Booting} \land \text{in} = \text{schedule} \rightarrow \]
\[ X(\text{schedules} = \{\text{data, business, search}\} \land \text{state} = \text{WaitToReplicate}) \]

Requirement 5: If the SyncClient is executing a replication job jobtype \(\in\) \{SR, BR, DR\} and is in state WaitToReplicate and receives a connection to a connection thread, the client will proceed to state WorkOnReplicate.

\[ G(\text{state} = \text{WaitToReplicate} \land \text{in} = \text{connectThread} \rightarrow (\text{jobtype} = \text{SR} \rightarrow \]
\[ X(\text{jobtype} = \text{SR} \land \text{state} = \text{WorkOnReplicate}) \land (\text{jobtype} = \text{BR} \rightarrow \]
\[ X(\text{jobtype} = \text{BR} \land \text{state} = \text{WorkOnReplicate}) \land (\text{jobtype} = \text{DR} \rightarrow \]
\[ X(\text{jobtype} = \text{DR} \land \text{state} = \text{WorkOnReplicate})) \]

Requirement 6: If the SyncClient is waiting either to replicate or boot and there is no more connection, the client proceeds to the End state.

\[ G(\text{state} \in \{\text{WaitToReplicate, WaitToBoot}\} \land \text{in} = \text{noConnectionThread} \rightarrow \]
\[ X(\text{state} = \text{End}) \]

Requirement 7: Once the SyncClient is in the End state, it cannot go to another different state.

\[ G(\text{state} = \text{End} \rightarrow X(\text{state} = \text{End})) \]

Requirement 8: If it is not in the End state then every schedule that the SyncClient possesses will eventually be executed as a replication job.

\[ G(\text{state} \neq \text{End} \rightarrow \text{search} \in \text{schedules} \rightarrow (F(\text{jobtype} = \text{SR}) \land \text{state} = \text{End}) \land \]
\[ \text{business} \in \text{schedules} \rightarrow (F(\text{jobtype} = \text{BR}) \land \text{state} = \text{End}) \land \]
\[ \text{data} \in \text{schedules} \rightarrow (F(\text{jobtype} = \text{DR}) \land \text{state} = \text{End})) \]

Requirement 9: The SyncClient cannot modify its underlying file system (files = readonly) unless it is in state WorkOnReplicate.

\[ G(\text{state} = \text{WorkOnReplicate} \rightarrow X(\text{files} = \text{writable} \land \text{state} \in \}
\[ \{\text{End, WaitToReplicate}\}) \land \text{state} \neq \text{WorkOnReplicate} \rightarrow \]
\[ X(\text{files} = \text{readonly} \land \text{state} = \text{WaitOnReplicate})) \]
**Requirement 10:** If the SyncClient is executing a replication job for a particular type of schedule, then that job can only receive schedules for that particular type of schedule.

\[ G(state = \text{WorkOnReplicate} \land in = \text{schedule} \rightarrow \]
\[ \quad \text{(search } \notin \text{ schedules } \land \text{ jobtype } = \text{SR} \rightarrow \]
\[ X(\text{search } \in \text{schedules}) \land (\text{business } \notin \text{schedules } \land \text{ jobtype } = \text{BR} \rightarrow \]
\[ X(\text{business } \in \text{schedules}) \land (\text{data } \notin \text{schedules } \land \text{ jobtype } = \text{DR} \rightarrow \]
\[ X(\text{data } \in \text{schedules})) \]

**Requirement 11:** If the SyncClient has committed to a schedule of a particular type and eventually that schedule is executed as a replication job then that schedule will be removed from the queue.

\[ G(\text{search } \in \text{schedules } \land \text{state } \in \{\text{WorkOnReplicate, Booting}\} \land \]
\[ F((\text{state } = \text{WaitToReplicate} \land \text{jobtype } = \text{nojob } \land in = \text{search job}) \rightarrow X(\text{jobtype } = \text{SR } \land \text{search } \notin \text{schedules})) \land F((\text{state } = \text{WaitToReplicate} \land \]
\[ \text{jobtype } = \text{nojob } \land in = \text{business job}) \rightarrow X(\text{jobtype } = \text{BR } \land \text{business } \notin \text{schedules})) \land \]
\[ F((\text{state } = \text{WaitToReplicate} \land \text{jobtype } = \text{nojob } \land in = \text{data job}) \rightarrow \]
\[ X(\text{jobtype } = \text{DR } \land \text{data } \notin \text{schedules})) \]

Table 5 gives the results obtained by running \textit{LBTest} on the 11 user requirements. For each requirement, we recorded the verdict (pass/fail/warning), the total time spent testing, the size of the learned hypothesis model at test termination, and the total number of model checker generated, learner generated and random test cases executed. To terminate each experiment, a maximum time bound of 5 hours was chosen. However, if the hypothesis model size had not changed over 10 consecutive random tests, then testing was terminated earlier than this.

### 4.1.1 Discussion of Errors Found

Nine out of eleven requirements were passed. For requirements 8 and 9, \textit{LBTest} gave warnings (due to a loop in the counterexample) corresponding to tests of liveness requirements that were never passed. The counterexample for both these requirements was “setAcceptorSchedulebusinessJobbusinessJob”. After the symbol “businessJob” a loop starts in the counterexample which has been unfolded just once. This violates requirement 8 because if we keep reading \textit{businessJob} from here the SUT does not go to the end state as specified. This violates requirement 9 because we are in the start state after reading this sequence rather than \textit{WaitOnReplicate} or \textit{End} states as specified. We do not reach any of these if we keep reading \textit{businessJob} from this state.
Table 5. Performance of LBTest on Fredhopper Access Server case study

<table>
<thead>
<tr>
<th>PLTL Requirement</th>
<th>Verdict</th>
<th>Total Testing Time (hours)</th>
<th>Hypothesis size (states)</th>
<th>Model checker queries</th>
<th>Learning queries</th>
<th>Random queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req 1</td>
<td>pass</td>
<td>5.0</td>
<td>8</td>
<td>0</td>
<td>50,897</td>
<td>45</td>
</tr>
<tr>
<td>Req 2</td>
<td>pass</td>
<td>5.0</td>
<td>15</td>
<td>2</td>
<td>49,226</td>
<td>13</td>
</tr>
<tr>
<td>Req 3</td>
<td>pass</td>
<td>1.7</td>
<td>11</td>
<td>0</td>
<td>16,543</td>
<td>17</td>
</tr>
<tr>
<td>Req 4</td>
<td>pass</td>
<td>2.1</td>
<td>11</td>
<td>0</td>
<td>20,114</td>
<td>14</td>
</tr>
<tr>
<td>Req 5</td>
<td>pass</td>
<td>2.5</td>
<td>11</td>
<td>0</td>
<td>24,944</td>
<td>17</td>
</tr>
<tr>
<td>Req 6</td>
<td>pass</td>
<td>2.3</td>
<td>11</td>
<td>0</td>
<td>23,215</td>
<td>16</td>
</tr>
<tr>
<td>Req 7</td>
<td>pass</td>
<td>2.1</td>
<td>11</td>
<td>0</td>
<td>18,287</td>
<td>17</td>
</tr>
<tr>
<td>Req 8</td>
<td>warning</td>
<td>1.9</td>
<td>8</td>
<td>15</td>
<td>18,263</td>
<td>12</td>
</tr>
<tr>
<td>Req 9</td>
<td>warning</td>
<td>3.8</td>
<td>15</td>
<td>18</td>
<td>35,831</td>
<td>18</td>
</tr>
<tr>
<td>Req 10</td>
<td>pass</td>
<td>2.7</td>
<td>11</td>
<td>0</td>
<td>26,596</td>
<td>19</td>
</tr>
<tr>
<td>Req 11</td>
<td>pass</td>
<td>4.6</td>
<td>11</td>
<td>0</td>
<td>45,937</td>
<td>21</td>
</tr>
</tbody>
</table>

A careful analysis of these requirements showed that both involved using the U (strong Until) operator. When this was replaced with a W (weak Until) operator no further warnings were seen for requirement 9. Therefore this was regarded as an error in the user requirements. However, LBTest continued to produce warnings for requirement 8, corresponding to a true SUT error. So in this case study LBTest functioned to uncover errors both in the user requirements and in the SUT.

4.2 Case Study 2: Brake-by-Wire

Our second case study consists of a brake-by-wire (BBW) system developed by Volvo Technology AB. A BBW system is an embedded vehicle application with ABS function, where no mechanical connection exists between the brake pedal and the brake actuators applied to the four wheels. A sensor attached to the brake pedal reads the pedal’s position percentage, which is used to compute the desired global brake torque. A software component distributes this global brake torque request to the four wheels. At each wheel, the ABS algorithm uses the corresponding brake torque request, the measured wheel speed, and the estimated vehicle speed to compute the actual brake torque on the wheel. For safety purposes, the ABS controller in the BBW system shall release the corresponding brake actuator when the slip rate of any wheel is larger than the threshold (e.g., 20%) and the vehicle is moving at a speed of above certain value, e.g., 10 km/h. A high level Simulink model of the BBW system is shown in Figure 3.

The BBW is a typical distributed system, which is realised by five ECUs (Electronic Control Units) connected via a network bus. The central ECU is attached with a brake pedal and an acceleration (gas) pedal. The other four
ECUs are connected to four wheels. The software components on the central ECU run the sensor of the brake pedal, calculation of the global brake torque from the brake pedal position and distribution of the global torque to the four wheels. The software components on each wheel ECU will measure the wheel speed, control the brake actuator and implement the ABS controller. The BBW is also a hard real-time system, i.e., it is specified with strict safety and temporal requirements, and runs 'continuously' by high frequency sampling (5-20 ms). The BBW has:

- two real-valued inputs: The inputs are received from the brake and gas pedals of the vehicle depending upon their positions and are denoted by breakPedalPos and gasPedalPos respectively. The position of the brake or gas pedal is bounded by the interval $[0.0, 100.0]$. 
- three real-valued outputs: the vehicle speed denoted by vehSpeed, rotational speeds of the four wheels of the vehicle (front right, front left, rear right and rear left) are denoted by $\omega_{SpeedFR}$, $\omega_{SpeedFL}$, $\omega_{SpeedRR}$ and $\omega_{SpeedRL}$ respectively. These speeds are bounded by the interval $[0.0, 111.0]$. Similarly the torque values on these wheels are denoted by torqueOnFR, torqueOnRL, torqueOnRR and torqueOnRL respectively the values of these torques are bounded by the interval $[0.0, 3000.0]$ nm.

The case study consisted of a Simulink model of a brake-by-wire (BBW) system developed by Volvo Technology. This model was translated into Java code consisting of about 1100 lines of code.

The infinite real valued data types can be addressed by defining discretisation formulas as described in Section 3.2. Therefore, an SUT wrapper is required to discretise the real-valued inputs and outputs of BBW into alphabets of finite symbols. The real-valued inputs are discretised into a set of four input events given by
\[ \Sigma_{in} = \{\text{brake}, \text{acc}, \text{accbrake}, \text{none}\} \]

Where the symbols \text{brake} and \text{acc} represent the conditions \text{brakePedalPos} = 100.0 and \text{accPedalPos} = 100.0 respectively, \text{accbrake} represents the condition \text{brakePedalPos} = 100.0 \land \text{accPedalPos} = 100.0 and similarly \text{none} represents \text{brakePedalPos} = 0.0 \land \text{accPedalPos} = 0.0 respectively. These symbolic input symbols are encoded with unicode characters as shown in Table 6 inside the \text{LBTest} tool.

<table>
<thead>
<tr>
<th>Constant Symbols</th>
<th>Encodings</th>
</tr>
</thead>
<tbody>
<tr>
<td>brake</td>
<td>a</td>
</tr>
<tr>
<td>acc</td>
<td>b</td>
</tr>
<tr>
<td>accbrake</td>
<td>c</td>
</tr>
<tr>
<td>none</td>
<td>d</td>
</tr>
</tbody>
</table>

Table 6. BBW Input Data Type

The discretisation formula for each symbolic output is also shown in column four of Table 7. Note that in Table 7, \text{vehSpeed}_i represents the vehicle speed at \text{i-th} event and hence the speed change at \text{i-th} event is \text{vehSpeed}_i - \text{vehSpeed}_{i-1}. The units of measurement for \text{vehSpeed}_i are \text{km/h} and the vehicle is considered as still at the \text{i-th} event if \text{vehSpeed}_i \leq 10 otherwise it is considered to be in motion. The vehicle is considered to be decelerating at the \text{i-th} event if \text{vehSpeed}_i < \text{vehSpeed}_{i-1} and \text{vehSpeed}_{i-1} > 0. The units of angular speed of the wheels are also converted into \text{km/h} inside the Java code of the SUT from the usual \text{rpm}. This is essential to calculate the slip rate of the wheels. The slip rate denoted by \text{slip} in Table 7 of a wheel (e.g. front right denoted by FR) is the ratio of the difference of \text{vehSpeed} - \text{wSpeedFR} and the vehicle speed \text{vehSpeed}. The vehicle is considered slipping when the slip rate \text{slip} > 0.2 otherwise the vehicle is considered not slipping.

After these discretizations of input and output values, three informal requirements can be formalized in PLTL as follows:

**Requirement 1:** If the brake pedal is pressed and the wheel speed (e.g., the front right wheel) is greater than zero, the value of brake torque enforced on the wheel by the corresponding ABS component will eventually be greater than 0.

\[ G(\text{in} = \text{brake} \rightarrow F(\text{wheel} = \text{wheelRotateFR} \rightarrow \text{torque} = \text{torqueOnFR})) \]

**Requirement 2:** If the brake pedal is pressed and the actual speed of the vehicle is larger than 10 km/h and the slippage sensor shows that a wheel is slipping, this implies that the corresponding brake torque at the wheel should be 0.

\[ G(\text{in} = \text{brake} \land \text{vehSpeed} > 10 \rightarrow F(\text{slip} > 0.2 \rightarrow \text{torque} = 0)) \]
<table>
<thead>
<tr>
<th>Data Type</th>
<th>Start Index</th>
<th>End Index</th>
<th>Discretisation Formula</th>
<th>Symbolic Output</th>
<th>Binary Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>speed</td>
<td>0</td>
<td>2</td>
<td>vehSpeed ≤ 10</td>
<td>vehicleStill</td>
<td>//, //</td>
</tr>
<tr>
<td>speed</td>
<td>0</td>
<td>2</td>
<td>vehSpeed &gt; 10</td>
<td>vehicleMove</td>
<td>//, t</td>
</tr>
<tr>
<td>speed</td>
<td>0</td>
<td>2</td>
<td>vehSpeed, &lt; vehSpeed_{i-1} ∧ vehSpeed_{i-1} &gt; 0</td>
<td>vehicleDecreased</td>
<td>t, //</td>
</tr>
<tr>
<td>wheel</td>
<td>2</td>
<td>4</td>
<td>ωSpeedFR &gt; 0</td>
<td>wheelRotateFR</td>
<td>//, //</td>
</tr>
<tr>
<td>wheel</td>
<td>2</td>
<td>4</td>
<td>ωSpeedFL &gt; 0</td>
<td>wheelRotateFL</td>
<td>//, t</td>
</tr>
<tr>
<td>wheel</td>
<td>2</td>
<td>4</td>
<td>ωSpeedRR &gt; 0</td>
<td>wheelRotateRR</td>
<td>t, //</td>
</tr>
<tr>
<td>wheel</td>
<td>2</td>
<td>4</td>
<td>ωSpeedRL &gt; 0</td>
<td>wheelRotateRL</td>
<td>t, t</td>
</tr>
<tr>
<td>torque</td>
<td>4</td>
<td>6</td>
<td>torqueOnFR &gt; 0</td>
<td>torqueFR</td>
<td>//, //</td>
</tr>
<tr>
<td>torque</td>
<td>4</td>
<td>6</td>
<td>torqueOnFL &gt; 0</td>
<td>torqueFL</td>
<td>//, t</td>
</tr>
<tr>
<td>torque</td>
<td>4</td>
<td>6</td>
<td>torqueOnRR &gt; 0</td>
<td>torqueRR</td>
<td>t, //</td>
</tr>
<tr>
<td>torque</td>
<td>4</td>
<td>6</td>
<td>torqueOnRL &gt; 0</td>
<td>torqueRL</td>
<td>t, t</td>
</tr>
<tr>
<td>slip</td>
<td>6</td>
<td>8</td>
<td>10 * (vehSpeed − ωSpeedFR) &gt; 2 * vehSpeed</td>
<td>slipFR</td>
<td>//, //</td>
</tr>
<tr>
<td>slip</td>
<td>6</td>
<td>8</td>
<td>10 * (vehSpeed − ωSpeedFL) &gt; 2 * vehSpeed</td>
<td>slipFL</td>
<td>//, t</td>
</tr>
<tr>
<td>slip</td>
<td>6</td>
<td>8</td>
<td>10 * (vehSpeed − ωSpeedRR) &gt; 2 * vehSpeed</td>
<td>slipRR</td>
<td>t, //</td>
</tr>
<tr>
<td>slip</td>
<td>6</td>
<td>8</td>
<td>10 * (vehSpeed − ωSpeedRL) &gt; 2 * vehSpeed</td>
<td>slipRL</td>
<td>t, t</td>
</tr>
</tbody>
</table>

Table 7. Brake by Wire Output Types
\[ G(\text{slip} = \text{slip}_FR \land \text{speed} = \text{vehicleMove} \land \text{in} = \text{brake}) \rightarrow \text{torque} \neq \text{torque}_OnFR \]

**Requirement 3:** If both the brake and gas pedals are pressed, the actual vehicle speed shall be decreased.

\[ G(\text{in} = \text{accelbrake} \rightarrow X(\text{speed} = \text{vehicleDecreased})) \]

<table>
<thead>
<tr>
<th>PLTL Requirement</th>
<th>Verdict</th>
<th>Total Testing Time (sec)</th>
<th>Hypothesis Size (States)</th>
<th>Model Checker Queries</th>
<th>Learning Queries</th>
<th>Random Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Req 1</td>
<td>Pass</td>
<td>2065</td>
<td>11</td>
<td>0</td>
<td>1501038</td>
<td>150</td>
</tr>
<tr>
<td>Req 2</td>
<td>Fail</td>
<td>58</td>
<td>537</td>
<td>18</td>
<td>34737</td>
<td>2</td>
</tr>
<tr>
<td>Req 3</td>
<td>Pass</td>
<td>960</td>
<td>22</td>
<td>0</td>
<td>1006275</td>
<td>130</td>
</tr>
</tbody>
</table>

Table 8. Performance of LBTest on Brake By Wire case study

4.2.1 Discussion of Errors Found When the BBW system was tested with the LBTest tool using the LTL requirements described above, requirements 1 and 3 were passed while LBTest continued to give errors for requirement 2 with different counterexamples during several testing sessions we tried with this requirement. The shortest counterexample found during these executions was “acc,acc,acc,acc,brake” which means that when the brake pedal is pressed after the vehicle has acquired a speed greater than 10 km/h and at that time when the slip rate of a wheel is greater than 20% then the SUT does not always have zero torque on the slipping wheel. All other counterexamples found also suggested a similar pattern of input sequences which when executed on the actual SUT confirmed that there was an error in the SUT corresponding to this requirement.

4.3 Conclusions from Case Studies

We have described the application of the LBTest tool to two case studies from different industrial domains. The tool successfully found errors in both of these case studies. This is despite the fact that one case study (the FAS) was operational for a relatively long time and has been thoroughly tested manually several times and these errors in it were not found earlier. The second case study (BBW) enforces strict timing constraints for safety reasons and also involves data from sensors that has to be suitably descretized for testing purposes. After the discretization LBTest revealed an error that violated a critical safety property to be enforced by the BBW on the vehicle. Furthermore the tool functioned to
find errors in the requirements as well for the FAS. The success of LBTest on industrial case studies reinforces our previous results on academic case studies and shows that LBT can be scaled to industrial problems.

5 Coverage

The notion of coverage metrics is well known in the software testing community. But nearly all metrics are for white box testing, where the tester has access to the source code of the SUT. The tester after executing test cases in a test suite can quantify the testing effort by using a metric that tells how much code has been covered by these test cases. However, the problem to define coverage metrics for requirements-based testing has scarcely been studied. It is nevertheless an important problem for the testing community that after any kind of testing activity, a tester has to quantify his/her achievements.

5.1 Requirements Coverage Metrics for LBTest

In [23] the authors have given some requirements coverage criteria for requirements-based testing. A test suite meeting each requirements coverage criteria is generated and the extent of model coverage achieved by it is evaluated on an executable model of the SUT. In the case of LBT there is no pre-existing model of the SUT available rather models are built and refined during each iteration of the LBT. Unlike in [23], LBT does not start with an existing test suite (generated on the basis of a requirements coverage criteria) rather test cases are constructed during execution. Therefore the requirements coverage criteria described in [23] cannot be used for LBTest before testing is started. The model coverage achieved by test cases generated by LBTest can however be measured as in [23] if we have access to the source code of an SUT which is to be tested against requirements using LBTest.

We chose as an SUT the simple cruise controller (CC) whose state transition diagram is shown in Figure 1. Java code replicating this state transition diagram was written and it was instrumented to record the following model coverage statistics about the test cases generated by LBTest:

1. State coverage,
2. Transition coverage,
3. Edge-pair coverage and
4. Prime path coverage.

Table 9 shows the coverage achieved by LBTest for each of the above mentioned criteria. The first column shows the coverage criteria used. Each experiment was repeated three times to take into account the varying influence of random test cases. The results for each experiment were recorded in three different rows against each coverage criteria. The second column shows the percentage of coverage achieved. The third column shows the minimum number of test cases (TCs) executed to reach the maximum coverage for each criteria and the last column shows the total number of test cases (TCs) executed for completely learning the SUT.
5.1.1 State Coverage Table 9 shows the state coverage achieved by LBTest in three different experiments. In the actual CC state diagram there are 8 states but the learning algorithm learns a minimal model of the system which is behaviourally equivalent to the actual system. We get 100% state coverage in this case for three different experiments because the random, model checker and active learning queries visit all the states of the SUT.

5.1.2 Transition Coverage The SUT consists of 8 states and since the input data type consists of five symbols therefore the total number of transitions for the SUT is 40. Table 9 also shows the transition coverage achieved by LBTest. This coverage as seen from the table is always 100% even before complete learning of the SUT as evident from the values in the third column of the table. This seems better than the results of [23] where transition coverage varies significantly across different requirements criteria.

5.1.3 Edge-Pair Coverage We did not compute the MC/DC coverage criteria because our code for the CC does not use any decision statements. We instead computed the edge-pair coverage (see [4]) achieved by LBTest on the CC example. For this purpose we wrote code to find out all the edge-pair paths in the CC example. The total number of edge-pair paths found were 81. But LBTest generated test cases as shown in Table 9 were not always able to achieve 100% edge-pair coverage. The reason seems to be that it is not necessary to visit every edge-pair to fully learn the SUT. So 100% edge-pair coverage contains redundant test cases when compared with LBT.

5.1.4 Prime Path Coverage Another coverage criteria given in [4] is prime path coverage. This is a rather strict coverage criteria as it does not allow any internal loops inside the path. Therefore the coverage for this particular metric

### Table 9. Coverage Metrics for LBTest

<table>
<thead>
<tr>
<th>Coverage Criteria</th>
<th>% Coverage</th>
<th>No. of TC\text{\textsubscript{max Cov}}</th>
<th>Total TCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>100</td>
<td>149</td>
<td>2437</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>17</td>
<td>2650</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>27</td>
<td>2424</td>
</tr>
<tr>
<td>Transition</td>
<td>100</td>
<td>797</td>
<td>4570</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>634</td>
<td>2740</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>112</td>
<td>2866</td>
</tr>
<tr>
<td>Edge Pair</td>
<td>100</td>
<td>813</td>
<td>2919</td>
</tr>
<tr>
<td></td>
<td>97.5</td>
<td>3370</td>
<td>3370</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>2096</td>
<td>2400</td>
</tr>
<tr>
<td>Prime Paths</td>
<td>44.8</td>
<td>2559</td>
<td>2559</td>
</tr>
<tr>
<td></td>
<td>44.8</td>
<td>2251</td>
<td>2251</td>
</tr>
<tr>
<td></td>
<td>51.7</td>
<td>2997</td>
<td>2997</td>
</tr>
</tbody>
</table>
is the lowest as compared to other metrics given in Table 9. We first wrote code
to find all prime paths in CC state diagram and the number of these paths was
found to be 87. But LBTest at best was able to cover roughly 50% of these
prime paths. The reason is the same as observed in the previous section that to
learn a behaviourally equivalent representation of the SUT it is not necessary to
cover all prime paths in the SUT. In this case we observe that 100% prime path
coverage has significant redundancy when compared with LBT.

6 Conclusions and Future Work

We have applied LBTest, a learning-based testing tool, to two industrial case
studies. In both case studies, the combination of computational learning methods
and model checking supported by LBT, was able to discover SUT errors which
had escaped previous extensive manual testing. These case studies illustrate the
scope and potential of LBT, as well as the practical difficulty of setting up
formal requirements testing in an industrial context. Above all, these industrial
case studies indicate the feasibility of applying LBT, with its high automation
and well established efficiency gains over other methods, to real world problems.

We have also presented some first results in measuring the coverage achieved
by LBT, albeit by using glass box coverage metrics to study our black-box testing
tool. Nevertheless, the results are interesting, not least because they suggest that
powerful learning algorithms can eliminate significant redundancy in some glass
box coverage models.

Further research needs to be devoted to understanding our combination of
partition testing and LBT which is needed for testing applications over infi-
nite data types. Alternatively, one can conduct LBT using more general model
checkers for full first-order linear temporal logic as in ([14]). It remains to be seen
which of these two competing approaches will ultimately be more successful for
tackling high-level data types.

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 and ARTEMIS project
269335 MBAT.

References

1. The conformiq designer tool (http://www.conformiq.com/products/conformiq-
designer/).
4. Paul Ammann and Jeff Offutt. Introduction to software testing. Cambridge Uni-
5. P. Chauhan, E. M. Clarke, J. H. Kukula, S. Sapra, H. Veith, and D. Wang. Au-
tomated abstraction refinement for model checking large state spaces using SAT
in Computer-Aided Design (FMCAD02), 2002.


