Learning-based Software Testing using Symbolic Constraint Solving Methods

FEI NIU

Licentiate Thesis
Stockholm, Sweden 2011
Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie licentiatexamen i datalogi måndagen den 7 november 2011 klockan 13.00 i F3, Lindstedtsvägen 26, Kungl Tekniska högskolan, Stockholm.

© Fei Niu, October 2011

Tryck: E-Print
Abstract

Software testing remains one of the most important but expensive approaches to ensure high-quality software today. In order to reduce the cost of testing, over the last several decades, various techniques such as formal verification and inductive learning have been used for test automation in previous research.

In this thesis, we present a specification-based black-box testing approach, *learning-based testing* (LBT), which is suitable for a wide range of systems, *e.g.* procedural and reactive systems. In the LBT architecture, given the requirement specification of a system under test (SUT), a large number of high-quality test cases can be iteratively generated, executed and evaluated by means of combining inductive learning with constraint solving.

We apply LBT to two types of systems, namely procedural and reactive systems. We specify a procedural system in Hoare logic and model it as a set of piecewise polynomials that can be locally and incrementally inferred. To automate test case generation (TCG), we use a quantifier elimination method, the Hoon-Collins cylindric algebraic decomposition (CAD), which is applied on only one local model (a bounded polynomial) at a time.

On the other hand, a reactive system is specified in temporal logic formulas, and modeled as an extended Mealy automaton over abstract data types (EMA) that can be incrementally learned as a complete term rewriting system (TRS) using the congruence generator extension (CGE) algorithm. We consider TCG for a reactive system as a bounded model checking problem, which can be further reformulated into a disunification problem and solved by narrowing.

The performance of the LBT frameworks is empirically evaluated against random testing for both procedural and reactive systems (executable models and programs). The results show that LBT is significantly more efficient than random testing in fault detection, *i.e.* less test cases and potentially less time are required than for random testing.
Acknowledgements

It is a pleasure to thank many people for their support, company and contributions to make this thesis possible.

Foremost, I would like to thank my supervisor, professor Karl Meinke. Throughout the past two and half years, his patience and kindness, as well as academic experience have been invaluable to me. Without his clever ideas, inspiring and classifying comments and questions on my work, this thesis would have never been finished. I also want to thank Mads Dam and Dilian Gurov for all the insightful advices, and Stefan Arnborg for reading and commenting on draft versions of this thesis.

I am indebted to all the theory group (TCS) members for providing a stimulating and fun environment in which to work and grow. Thank you to Emma Enström, Siavash Soleimanifard, Pedro de Carvalho Gomes, Musard Balliu, Lukáš Poláček, Sangxia Huang and Karl Palmskog for great time we spend in the many lunches and football games. I am grateful to my office mates, Andreas Lundblad and Muddassar Azam Sindhu for giving me wise advices and warm help.

I wish to thank all my Chinese friends in Stockholm, especially: Qin Yuchu, Liu Miao, Mao bo, Li Yuhen, Rui Yikang, Gao Zhan and Niu Xin, for all the great time we spend together.

I gratefully acknowledge financial support for the work presented here from the Chinese Scholarship Council (CSC), the Swedish Research Council (VR) and the European Union under project HATS FP7-231620.

Finally, I thank my parents, Niu Hongyan and Yang Cuiying, and my girlfriend, Dong Jing, for your profound love and support to me in all aspects of my life.
Chapter 1

Introduction

With the increasing complexity and importance of software systems, further improvement of techniques for verifying high-quality software, such as testing and formal methods, are urgently needed. Software testing is still vitally important in the software development process as the use of formal verification techniques remains limited today.

This thesis concerns an iterative testing approach for procedural and reactive systems, called learning-based testing. It is a specification-based black-box testing method that encompasses both test case generation, execution and evaluation (the oracle step) by combining inductive learning, formal methods and constraint solving techniques. To provide a context for our research work, we will review related research in literature, including various related black-box testing methods involving inductive learning and model checking techniques.

This thesis is organised as follows. In the remainder of Chapter 1, we describe the background to our research which is software testing. We give a simple overview of our research area, learning-based testing for procedural and reactive systems. In Chapter 2, we discuss black-box testing and several common testing notions such as test selection and adequacy criteria, requirements specification languages and major testing strategies. A comparison of our approach in terms of specification-based testing and other approaches to testing are given in Section 2.3. Chapter 3 discusses inductive learning, its application in software testing and incremental learning techniques in learning-based testing. Constraint solving techniques are summarised in Chapter 4 and we show how test cases can be generated by constraint solving and bounded model checking in LBT and review test case generation using model checking in literature. Two appendices at the end of this thesis includes two published papers:

CHAPTER 1. INTRODUCTION


1.1 Background

Software testing today remains the most important approach for software quality assurance in industry. A recent detailed survey in the United States [30] estimates that testing may cost up to 30-50% of a project budget and shows the high economic impact of the improved software testing technology, i.e., achievable savings from improved testing method are estimated to be $22.2 billion or 0.2% of US GDP.

By exercising a finite number of test cases, software testing aims to observe executions of an implementation of a system under test (SUT) to validate whether it behaves as expected and identify potential faults. It is well recognised that formal specification and verification techniques can be used as the basis to automate test case generation and test verdicts in order to improve the efficiency of software testing. One example of this approach is specification-based testing [72], where an SUT is tested against a set of formally defined requirements. Requirements usually involve both input and output of the SUT: test cases are generated from requirements and the test results are expected to conform to requirements or else a test is failed. Note that requirements can be used describe any property of the SUT such as functionality or performance, but in this thesis we focus on functional requirements, i.e., functional testing.

There are two important test design methods termed glass-box (or white-box) and black-box testing [8]. Glass-box testing uses internal knowledge of the SUT to guide the selection of test data while black-box testing ignores this internal structure. Glass-box methods typically scale exponentially with system size and complexity and it is much more difficult to maintain regression testing when the SUT evolves during the development process. Black-box testing only focuses on functional requirements, which can make it more scalable and efficient than glass-box testing.

Another important recently development in testing is model-based testing (e.g. [76, 22]). With the introduction of manually created models, test cases can be constructed manually. However, with efficient model checkers test cases can also be constructed automatically (e.g. [28]). Nevertheless, a complete and precisely defined model is rarely available in practise due to the difficult and high cost of the model creation process. This makes machine learning for model extraction an interesting problem that is closely related to our research (see [1]).
1.2. Overview

The key idea of learning-based testing is that, for a specific type of system, we choose a suitable formal model that can be inductively learned through interacting with the system so that given functional requirements can be algorithmically checked against the learned model, and counterexamples (if any) are constructed as relevant test cases.

As shown in Figure 1.1, learning-based testing is based on three important components: a system under test $S$, a formal requirements specification $\text{Req}$ and a learned model $M$ of $S$. In Figure 1.1 the learning algorithm provides a feedback loop from the test execution and evaluation back to test case generation (TCG). An abstract but more detailed description of LBT can be given in terms of the following four steps:

1. Suppose that $n$ test cases have been executed on $S$, which means the current observation set of $S$ is a set of input/output pairs: $\{(i_1, o_1), \ldots, (i_n, o_n)\}$. A model $M_n$ is learned based on the current observation set. For $n \geq 2$, $M_n$ is generated by refining $M_{n-1}$. Note that the learned model is approximately correct (before $S$ is completely learned) and provides the basis for us to predict unknown faults hidden in the SUT, by the process of generalisation from the observed data.

2. By a decision procedure, we can determine whether any violation of $\text{Req}$ (or satisfaction of $\lnot \text{Req}$) on $M_n$ exists. If a violation exists, an integrated constraint solving algorithm returns a counterexample to be instantiated into a test case $i_{n+1}$, which illustrates how $\text{Req}$ is violated. Otherwise, $i_{n+1}$ is randomly chosen or generated to be favourable to the refinement of $M$.

3. Run the test case $i_{n+1}$ on the SUT and record the output $o_{n+1}$. If $S$ failed $(i_{n+1}, o_{n+1})$ (e.g. $(i_{n+1}, o_{n+1})$ falsify $\text{Req}$), it means $i_{n+1}$ is a true negative and we jump to step 4, else $S$ passes $(i_{n+1}, o_{n+1})$ (this is possible since $M$,
is inaccurate), and \( i_{n+1} \) is a false negative. In this case, the current iteration finishes here. However, with the presence of the feedback path from Oracle to Learner in Figure 1.1, the false negative test case is not a waste of testing effort: In step 1 of the next iteration, the observation of \( S \) is updated with \( t_{n+1} \) and \( M_i \) is refined to be \( M_{i+1} \) that is more accurate.

4. The test terminates and the test case \((i_{n+1}, o_{n+1})\) is provided as one witness of an underlying fault in \( S \).

The actual implementation of these steps will vary according to different requirements languages, system models, learning algorithms and constraint solving techniques. So more detailed accounts will be found in Papers I and II.

1.2.1 Modelling Systems Under Test

It is useful to classify different kinds of SUTs since they have distinct properties that fundamentally affect the way testing must be carried out. In this thesis we focus on two important types of software system, namely:

1. procedural systems, and
2. reactive systems.

As shown in Figure 1.2a a procedural program computes from an initial input data \( i \in I \) in finite time (if it terminates) to a final output data \( o \in O \). This kind of system can be naturally modelled as a black-box implementing an unknown function \( f : I \rightarrow O \). Typical examples are C functions, Java methods and numerical procedures.

The term reactive system introduced in [38] is now commonly used to designate continuously operating systems. Unlike a procedural system, a reactive system normally does not terminate but rather maintains an ongoing interaction with the environment, i.e. it accepts an input from the environment and produces an output to the environment. Typical examples of reactive systems are embedded control systems of automobiles, trains, planes and so on.

In Figure 1.2b, a black-box model of a reactive system can be conceived as a function \( F : (T \rightarrow I) \rightarrow (T \rightarrow O) \), where \( T \) is a set of time points when inputs and outputs occur. For \( T = \mathbb{N} \), the input and output of the reactive system are
1.2. OVERVIEW

time-ordered sequences of discrete events/actions (i.e. \(i_0 i_1 i_2 \ldots\) and \(o_0 o_1 o_2 \ldots\) in Figure 1.2b). In the case of \(T = \mathbb{R}\), the input and output of the reactive system become continuous time-varying values. Examples of such reactive systems are hybrid systems [2]. In this thesis, we consider discrete time only, which is normally modelled as a Mealy automaton.

**Definition 1.2.1 (Mealy Automaton)** A Mealy automaton is a 6-tuple \((S, S_0, \Sigma, \Lambda, \sigma, \lambda)\), consisting of the following:

- \(S\) - a finite set of states.
- \(S_0\) - the initial state, which is an element of \(S\).
- \(\Sigma\) - the input alphabet.
- \(\Lambda\) - the output alphabet.
- \(\delta\) - the transition function mapping pairs of a state and an input symbol to the corresponding next state.
- \(\lambda\) - the output function mapping pairs of a state and an input symbol to the corresponding output symbol.

Given a Mealy automaton \(M\), there is a corresponding set \(L(M)\) of behaviours, the set of input/output sequences (traces) that can occur from the initial state of \(M\). This generalises the concept of a language acceptor.

1.2.2 Testing from Requirements

An important feature of learning-based testing is that we test from a set of functional requirements. Functional requirements define what an SUT is supposed to do normally from point of view of a user, and they are essentially a set of (positive or negative) constraints on the input/output behaviour of a system. Non-functional requirements cover various quality attributes such as security, usability, maintainability, extensibility and scalability.

As suggested in the previous section, functional requirements are dependent upon the type of SUT and thus are described in different specification languages. For procedural systems, we use a first order language such as Hoare logic [40]. For reactive systems, due to the notion of time, we usually choose temporal Logic [24], for example, Linear Temporal logic (LTL) or Computational Tree logic (CTL or CTL*).
Chapter 2

Black-box Testing

Black-box testing is an important collection of testing methods that represent 35% to 65% of all testing [8]. In the presence of well defined formal system requirements and system models, black-box testing dominates since it is much more scalable and efficient than other testing techniques (e.g. structural testing).

In this chapter, we give an introduction to software testing and in particular black-box testing. LBT [58] can be considered a novel specification-based black-box testing method that combines inductive learning and model checking techniques.

2.1 Software Testing

What is software testing? A brief description of software testing is given in [11] as follows: "Software testing consists of the dynamic verification of the behaviour of a program on a finite set of test cases, suitably selected from the usually infinite execution domain, against the expected behaviour."

In the description above, there are four main features of testing that we would like to highlight here.

- Dynamic. Software testing requires the execution of the SUT with specific input values. One of the big advantages over static techniques such as symbolic execution and static program analysis is that testing is able to provide information about an SUT's behaviour, and verify both the SUT and its execution environment, for example, the compiler, external libraries, OS, network I/O and so on.

- Finite and selected. Exhaustive testing is impossible in the available time since the number of possible tests is usually large or infinitely many. For reactive systems, the length of an input sequence should also be finite.

- Expected behaviour. Software is expected to exhibit certain behaviour or fulfil certain requirements. After each test execution, the oracle has to decide whether the SUT has reached a failure (undesired behaviour) or not.
CHAPTER 2. BLACK-BOX TESTING

There are many different kinds of testing classified in different ways. One common kind of testing technique is functional testing, where testers are only concerned about the functionality of the SUT. Performance testing tests non-functional requirements such as the throughput of the system under heavy load, and robustness testing is focused on the robustness and fault tolerance of software under unexpected environments. In another dimension, testing can be classified as black-box and glass-box/white-box (or structural) methods with respect to the kind of information we use to design tests. White-box (glass-box) tests are derived from the (complete) structural information of the SUT, while black-box testing means that we only use system requirements specification that describe the expected behaviour of the SUT, treating the SUT as a black-box.

2.1.1 Test Selection and Adequacy Criteria

Test selection and adequacy are two complementary notions in testing. Test selection is about how we test and test adequacy is about how much we should test. Several important test selection and adequacy criteria are introduced as follows:

- **Coverage criteria.** Most testing techniques are designed and assessed based on certain kinds of coverage criteria. The coverage measurement typically refers to code properties, such as function coverage, statement coverage, condition coverage, decision coverage and so on. Similarly, in model-based testing, structural coverage can also be measured in terms of the model’s structure [76, 74].

- **Fault-based criteria.** The faults of the coverage analysis approach are obvious. The goal of testing is undoubtedly to detect faults hidden in the SUT rather than to fulfill coverage criteria. Fault-based criteria have been proposed in [61] to emphasise the fault detection nature of testing techniques. A typical example of fault-based criterion is mutation testing, where programs are mutated by apply a set of mutation operations to contain some faults, typically frequently occurring faults and the quality of a test suite can be measured by how many faults this test suite is able to detect.

- **Inference-based adequacy criteria.** Inference-based adequacy criteria [80, 82, 78] refers to a special coverage criteria that attempts to measure test adequacy based on a behavioural model inferred from input/output data of an SUT. A test set is considered to be adequate if the inferred model is equivalent (or approximately equivalent) to the SUT. Similarly in LBT, an equivalence checker is used to compare the learned model with the SUT and stops testing when the learned model is equivalent (or approximately equivalent) to the SUT.
2.1. SOFTWARE TESTING

2.1.2 Requirements Specification Languages

With a formal functional requirements specification that is precise and unambiguous, one can either prove the correctness of an SUT (formal verification), or derive test cases automatically for its black-box description. There are a great variety of different methods to specify the functional requirements of an SUT, which can be classified as follows:

- **State-based (or pre/post condition) specification.** These specify how the internal state of an SUT is changed by function (or method) calls and are usually described in terms of pre and postconditions on a function (or method). Typical examples include Z, VDM, Java Modelling Language (JML) [49], Spec# [73], the UML Object Constraint Language (OCL) [79] etc. In Paper I, we will use a Hoare triple (see e.g. [51]), as a pre/postcondition specification for a procedural system $S$ in LBT in the form of $\{\text{pre}\} \ S \ \{\text{post}\}$. Both pre and post are written in the language of first-order logic over real closed fields [71]. Under the usual partial correctness interpretation of Hoare triples (no assumption of termination), any test case that satisfies the formula $\text{pre} \wedge \neg \text{post}$ is considered to be failed by $S$.

- **Trace-based specification.** In this case, the behaviour of an SUT is specified with respect to the history of allowable traces (input/output sequences) over time. As suggested in Section 1.2, various notions of time can be used (discrete or continuous, linear or branching etc.), leading to many kinds of temporal logic languages. In LBT, we choose LTL as the specification language for reactive systems.

- **Algebraic specification.** This describes a system as a collection of mathematical functions over abstract data types (ADTs). It encapsulates data and operations that manipulate that data. For example, the property $\text{pop}(\text{push}(s, x)) = s$ specifies that the pop operation ($\text{Stack} \rightarrow \text{Stack}$) undoes the effect of a push operation ($\text{Stack} \times \text{Element} \rightarrow \text{Stack}$) on a stack $s$ with an element $x$. Examples of algebraic specification language include OBJ [31] and the Common Algebraic Specification Language (CASL) [62]. Moreover, there is a special kind of algebraic specification *process algebra*, which describes a system as a number of communicating concurrent processes based on universal algebra. Examples of process algebras include CSP [41], CCS [60] and and so on. The similarity between algebraic specifications of ADTs and class interfaces suggests that there is an important potential role for formal algebraic specifications in testing Object-Oriented (OO) software [39].

- **Visual transition specification.** This kind of specification describes the transitions between two different states of an SUT using graphical representation. For example, in a visual model, such as a finite state machine model (FSMs), we have a graph where the nodes represent the states and arcs represent the
transitions including inputs and/or outputs (i.e. Mealy machines or Moore machines). In industry, such languages are extended to be even more expressive by adding predicates or context (or global) variables on transitions, (abstract) data types and hierarchies, known as extended FSMs (EFSMs) [69]. Various statecharts such as UML State Machines, STATEMATE statecharts and Simulink Stateflow charts are widely used tools of transition-based specification and models.

2.2 Strategies for Black-box Testing

A test strategy refers to a systematic method for selecting or generating test cases to be included in a test suite (and possibly constructing the oracle).

Without access to the internal structure of an SUT, black-box testing approaches mainly rely on requirements specifications. However, there are two kinds of black-box testing methods, random testing and inductive testing, where requirements specifications are not required for test case generation.

2.2.1 Random Testing without a Strategy

Random testing is the opposite of systematic method. The input domain is defined, data points (test cases) are selected randomly and independently from this domain and these are executed on an SUT. The test results are hence evaluated according to the requirement specification.

Despite the lack of a systematic strategy, random testing is still a desirable testing method according to [37] because:

1. there are efficient methods for random data selection by computing pseudorandom numbers or low-discrepancy sequences used in the Quasi-Monte Carlo method [64]. A large number of test cases can be easily and quickly generated. The ability of generating an unlimited number of test cases fairly quickly makes random testing a good benchmark for evaluating LBT.

2. The randomness in test case generation allows the method to be used for statistical assessment and predication of system reliability, e.g., it is 99% certain that an SUT will fail no more than once in 10000 runs [37].

2.2.2 Other Traditional Black-box Strategies

Traditionally two frequently used strategies are equivalence partitioning and boundary value analysis [47]. According to the input conditions in a requirements specification, the equivalence partitioning technique divides the input domain data of an SUT into partitions of valid or invalid inputs and chooses at least one test case from each partition class. For example, if an input condition of an SUT component specifies a range, one valid and two invalid equivalence classes are defined. Boundary value analysis extends equivalence partitioning by considering the boundary
values of equivalence classes. This technique is based on the fact that faults tend
to occur at the boundaries of the input domain rather than in the centre.

Pairwise testing [63] is a combinatorial software testing method, which mainly
aims to achieve dimension reduction and is typically used for procedural systems.
For each pair of input variables, pairwise testing tests all possible discrete pair com-
binations of those values with a fixed assignment to the remaining variables. More
generally, \( n \)-wise testing refers to the combination of \( n \) input variables. Pairwise
testing can be combined with LBT for procedural systems as we show in Paper I.

2.2.3 Specification-based and Model-based Strategies

In specification-based testing (see e.g. [72]), a well-defined specification becomes the
definitive reference for software testing. Since conditions, states and transitions of
an SUT may appear in the requirements specification, it is possible to apply various
coverage criteria to certain kinds of specifications. For example, for transition-based
specifications, one can design test sets that exercise every transition at least once
(all-transitions coverage).

What makes specification-based testing a promising test method is that test
cases are generated from a functional requirements specification to include verdict
information. This means that it is possible to determine accurately whether a test
case has been passed or failed by an SUT by checking the observed behaviour or
output against that specification (instead of simply observing system crashes or
exceptions, which is quite unsatisfactory).

Model-based testing (MBT) [76] has been used to name a wide variety of test case
generation techniques in recent years. The term model refers to different concepts,
e.g.:

- Abstract test cases (a model of test sets), from which executable test scripts
can be instantiated. One example would be transforming a sequence diagram
in UML into low-level test scripts.
- Various behavioural models such as JML, OCL and statecharts. In this case, a
model-based testing tool can generate test cases to fulfil coverage criteria such
as all-transitions coverage or all-states coverage on a statecharts model, or all-
branch coverage on a OCL model. In addition, some test verdict information
is included as these behavioural models contain the relation between inputs
and outputs.
- Various system models that do not involve the relation between inputs and
outputs, such as data domain models, environment/usage models.

2.2.4 Inductive Testing

Inductive testing introduces inductive learning (or model inference) into software
testing. This method can be applied even in the absence of a requirements speci-
fication of an SUT. Earlier research on inductive testing (e.g. [82, 81]) focused on
CHAPTER 2. BLACK-BOX TESTING

Test adequacy measurement using inductive learning: it used existing tests to infer a model or program, and the test set could be considered to be adequate if the model is shown to be equivalent to the underlying program. More modern research has become more flexible and practical in the sense that an inferred model is not required to be exactly equivalent to the SUT [77] and can be used for test case generation [78].

2.3 Learning-based Testing

In this section, we will: (i) compare and contrast LBT with the traditional and more recent black-box testing techniques that have been summarised in Section 2.2; and (ii) identify specific contributions of our research to the literature.

We consider LBT to be a specification-based black-box approach to testing that utilises: (a) a formal requirements specification, (b) constraint solving and (c) inductive learning techniques. According to the specific type of SUT (i.e. procedural or reactive systems), we apply specific requirements languages, learning algorithms and constraint solving techniques to achieve automation of test case generation, test case execution and evaluation. The main differences (and similarities) between LBT and other testing approaches are as follows:

- Both LBT and inductive testing use inductive learning techniques. However, inductive testing focuses on specification-free testing, while LBT is a specification-based testing method which makes heavy use of formal requirements specification and formal verification techniques (e.g. model checking and constraint solving).

- A behavioural model in MBT is manually constructed as a part of model-based development, and is expensive to create and update. In LBT, on the other hand, a behavioural model is automatically inferred and updated using observed behaviour data from testing, which is also known as reverse engineering. Compared to model-based testing, model construction and updating are much cheaper in LBT.

- Both LBT and MBT use constraint solving to generate test cases. i.e., computing solutions for the free variables in a negated specification over a model. In MBT, TCG is often considered as a model checking problem that can be generalised into a constraint solving problem.

- Test objectives concern why we test and what we hope to accomplish. The test objective of LBT is to explore a requirement violation based on a requirements specification as a functional constraint. However, in MBT of reactive systems, a specification (in temporal logic) is essentially a structural constraint and its test objective is mainly about satisfying a coverage criterion.

Several previous studies such as black-box checking [68] and its variant, adaptive model checking [34] have considered a combination of learning strategies [67]
and model checking to achieve testing and/or formal verification of reactive systems. Within the verification community, an iterative model checking techniques, known as *counterexample guided abstraction refinement* (CEGAR) [17] also applies machine learning techniques to abstraction and abstraction refinement (see e.g. [18] and [35]). However LBT can be distinguished from those techniques by the following characteristics:

- LBT focuses on testing rather than formal verification and doesn’t involve abstraction functions (e.g. on data) that are constructed and refined based on learning in CEGAR. However, incremental learning algorithms are in some sense an abstraction method, too.

- LBT uses incremental learning algorithms that are more efficient, scalable and suitable for testing, while black-box checking and adaptive model checking use complete learning algorithms (c.f. section 3.1).
Chapter 3

Inductive Learning

In this chapter we introduce inductive learning techniques and survey various automata learning algorithms in the literature. As suggested in the previous chapter, test results, i.e. input/output pairs, are observations and verdicts about a black-box SUT. Based on these observations and verdicts it is possible to inductively generalise a behavioural model, which can be useful for program analysis or test case generation.

3.1 Inductive Learning Functions and Languages

Inductive learning is an important type of machine learning. It can be considered as a possibly infinite process to hypothesise a general system description or rule set, usually a function or language expression, from samples of a target object or system. There are important applications of inductive learning in a variety of different domains, including language acquisition, pattern recognition, bio-informatics, intelligent agent design and software reverse-engineering/dynamic analysis.

3.1.1 Learning in the Limit

Gold’s theoretical study of language identification [32] established the first formal model of inductive learning as learning in the limit. Since then more and more models have been developed based on that (see the survey [5]). Learning in the limit is a quite general paradigm of sequential learning. Abstractly speaking, suppose that $L$ is an inductive learning method that is expected to learn correctly an unknown target $T$. Suppose that $L$ iteratively generates a sequence $R_1, R_2, \ldots$ of rule sets which describe the models $M_1, M_2, \ldots$ of the target $T$. If at some point the models converge, which means there exists some number $n$ such that $M_n$ agrees with the current set of examples of $T$ and

$$M_n \sqsubseteq M_{n+1} \sqsubseteq M_{n+2} \sqsubseteq \ldots,$$

15
then $L$ learns $T$ correctly in the limit, although $L$ may not be able to determine whether it has reached a correct hypothesis (behavioural equivalence). Note that in most cases it is realistic to weaken the definition of learning in the limit by considering hypotheses that are approximately correct. This allows more systems to be effectively learnable.

### 3.1.2 Generalised Incremental Learning

The learning techniques (piecewise polynomial function approximation and the CGE algorithm of [57]) used in Papers I and II are able to learn in the limit as defined above. More importantly, these are incremental learning approaches. Incremental learning is normally used in scenarios where the observation data is sequentially supplied, and involves an iterative process of generating a new hypothesis refined from the previous one whenever any new data is available. On the other hand, complete learning pursues the goal of learning the complete target object from a set of examples and checking behavioural equivalence explicitly and repeatedly.

In the context of testing, incremental learning algorithms are considered to be more efficient and scalable for the following reasons:

- Real reactive systems are normally too big in terms of state space size to be completely learned in a reasonable amount of time, and testing specific requirements such as use cases may not require learning the entire state space of the reactive systems.

- The learner requires little or no specific data before constructing a hypothesis. This allows all (or more of) test cases to be determined by a model checking process.

- In cases where the complexity of learning algorithms grows quadratically or worse in the size of target systems, the incremental learning approach can still efficiently construct hypotheses before reaching the ‘elbow’ of the curve.

- Adding and refining local models is one way to achieve incremental learning (c.f. Section 3.2), which is used in Paper I for numerical programs.

### 3.2 Piecewise Polynomial Approximation

In Paper I, we consider how to model a procedural SUT by means of a function $f : A \rightarrow B$. More specifically, we model a numerical program as a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ that is unpacked in terms of the number of outputs to $f_1, f_2, \ldots, f_m : \mathbb{R}^n \rightarrow \mathbb{R}$. The function $f_i$ is learned as a set $\{p_1^i, p_2^i, \ldots, p_k^i\}$ of overlapping local models: piecewise polynomials, where $p_j^i$ is an $n$-dimensional and $d$-degree polynomial defined over an $n$-dimensional sphere of radius $r$ over the input/output space. Since $(d+1)^n$ points suffice to uniquely determine an $n$-dimensional degree $d$ polynomial,
each local model has \((d + 1)^n\) such points. One such point \(c \in \mathbb{R}^n\) is distinguished as the centre point. The radius \(r\) of a local model is the maximum of the Euclidean distances from the centre point to each other point in that model. With \(n\)-dimensional polynomials we can automate \(k\)-wise testing where \(k < n\), e.g. for \(k = 2\) we obtain pairwise testing.

Indeed, inductively learning a numerical program concerns approximating polynomial functions of real numbers (or curve fitting). By the Weierstrass Approximation Theorem (see e.g. [36]), there always exists a sequence of polynomials which converges to any given continuous function on a compact subset of \(\mathbb{R}^n\). This implies that, for any numerical program \(P\), we are able to learn it in the limit in the sense that each local model (a polynomial) we learned eventually converges to the underlying function \(f\) of \(P\) over the domain of the local model, and undoubtedly satisfies the pre and postconditions of the numerical program.

Also in Paper I, a numerical program is incrementally learned in the sense that each time any new data is obtained, only those local models that are affected by the new data are updated. Whenever a local model is updated, we use a Monte Carlo method to efficiently approximate its convergence value. A numerical program can be any degree of complexity over the entire input domain, but is like to contain simple behaviours (or curves) over certain sub-domains. By learning a numerical program as a set of local models, the incremental learning approach is able to identify simple curve fragments by measuring convergence and construct converged local models quickly. Intuitively, test cases generated from better converged models are of better quality.

### 3.3 Grammar Inference and Automata Learning

Grammatical inference is a special case of inductive learning, which attempts to learn a formal grammar \(G\) or an equivalent machine that recognises a language \(L(G)\) from a set of positive or/and negative strings. The simplest class of languages that can be learned are regular languages produced by regular grammars. A regular grammar can be represented as deterministic finite state automata (or state machine) and most work on regular grammar inference uses this kind of representation as it is simple and well-studied [66, 21].

**Definition 3.3.1 (Deterministic Finite Automaton)** A deterministic finite automaton (DFA) is a tuple \(A = (Q, \Sigma, \delta, q_0, F)\), where \(Q\) is a finite set of states, \(\Sigma\) is a finite alphabet, \(\delta\) is the transition function \(Q \times \Sigma \rightarrow Q\), \(q_0 \in Q\) is the initial state and \(T \subseteq Q\) is the set of accepting states. If \(w = a_1, a_2, \ldots, a_n\) is a string over the alphabet \(\Sigma\), the automaton \(A\) accepts the string \(w\) if there exists a sequence of states, \(s_0, s_1, \ldots, s_n\) such that \(s_0 = q_0, s_n = F, s_{i+1} = \delta(s_i, a_{i+1}), i \in [0, n)\).

Given the definition of DFA, the regular grammatical inference problem can be considered as identifying a DFA, \(A(L)\) for a language \(L\) from a set of example
strings $S$, where $S = S^+ \cup S^-$ and $S^+ \cap S^- = \emptyset$, such that $A$ accepts all the positive examples $S^+$ and rejects all the negative examples $S^-$.

### 3.3.1 State Merging and Automata Minimisation

Although there can be many DFAs that accept a given regular language, the Myhill-Nerode theorem [45] states that the DFA that accepts a regular language using a minimal number of states is unique. This DFA is called the minimal or canonical DFA for the language. Often it is required that grammar inference algorithms must return a minimal DFA and in any case there are well established techniques in [45] that can be used to derive a minimal equivalent automaton from a given non-minimal automaton.

State merging is the basic idea for automata learning and minimisation originally developed in [75]. To infer a minimal DFA representing the target grammar, state merging is repeatedly performed on an initial model, typically an augmented prefix tree acceptor (APTA), a tree-shaped state machine that literally represents positive and negative sequences in the given set of examples.

An automata learning or grammar inference process iteratively involves (i) identifying an equivalent pair of states by comparing their suffix-tree, and (ii) merging these two states until the APTA converges upon a minimal DFA of the target grammar. Figure 3.1 shows a state merge step on an ATPA that represents $S^+ = \{a, ac, aca, bc, bea, c\}$ and $S^- = \{acb, b, bcb\}$.

![Figure 3.1: One State Merge Step on an APTA](image)

### 3.3.2 Active Learning and Incremental Learning of Automata

Gold’s initial research [33] has shown that finding a minimum DFA from a given finite set of positive and negative examples is NP-hard in general. However, a number of learning algorithms are proposed to learn a minimal DFA in polynomial time, by adding an oracle that provides extra information for learning algorithms or restricting the training data.

In active learning approaches, the learner is able to actively choose any string to be labelled by a target grammar, and hence an active learning process is normally iterative and the sample set grows over time. The most notable example of
active automata learning could be Angluin’s $L^*$ algorithm, where a DFA is always exactly learned in polynomial time with the help of a minimally adequate teacher (MAT) and an oracle. The MAT is capable of answering membership queries with a true or false response and the oracle answers equivalence queries with a yes or a counterexample. Passive learning techniques, the counterpart of active learning, synthesise an automaton from a given and fixed set of samples. Biermann’s $k$-tail algorithm and [10] and the regular positive and negative inference (RPNI) algorithm by Oncina and Garcia [65] and Lang [48] are well-known examples of passive learning. Without a teacher, RPNI can produce an accurate model, provided that $S$ contains a characteristic sample of the target grammar. Normally, an active learning algorithm requires a certain number of membership queries to accomplish internal book-keeping (e.g. a closed, consistent observation table in $L^*$) before generating any hypothesis. For complex automata, the number of required membership queries becomes unacceptably large and the learning is expensive in terms of computation time.

In the context of DFA learning, incremental learning refers to the iterative and incremental automata construction techniques for sequential presentation of data. Thus strings are sequentially fed into a learner, and whenever any amount of new data is available the learner is able to produce a new hypothesis automaton based on the current one in response. There are two cases for automata construction: (i) if the new data is consistent with the current hypothesis automaton, it remains the same, or else (ii) part or the whole of the current hypothesis automaton has to be changed to conform with all the observed strings. Typical examples of incremental automata learning are RPNI2 [23] (an incremental version of RPNI) and CGE [57].
Chapter 4

Constraint Solving for Test Case Generation

A satisfiability problem concerns the question: is a formula \( \phi \) true over a model (or structure) \( M \) for some assignment \( \alpha \)?, which is denoted by \( M, \alpha \models \phi \). Constraint solving concerns procedures to solve this existential problem by computing satisfying assignments \( \alpha : X \rightarrow M \) for the free variables in \( \phi \). In the application to TCG, \( \phi \) is typically the negated user requirements specification expressed in certain kind of logic, \( M \) captures system behaviours and the solutions (assignments to \( X \)) are interpreted as test cases that violate the requirements. For procedural systems, \( \phi \) can be expressed in first-order logic and \( M \) is a first-order structure (such as a real closed field in Paper I). For reactive systems, \( \phi \) can be expressed in temporal logic and \( M \) refers to extended Mealy automata (in Paper II).

We therefore use constraint solving techniques to automate TCG. For numerical programs, TCG can be based on Tarski’s quantifier elimination method [14] to solve first order formulas over real closed fields. For reactive systems, TCG is automated by bounded model checking that can be reformulated as a disunification problem and solved by term rewriting techniques, especially narrowing. These two specific constraint solving techniques are discussed in Sections 4.1 and 4.2 respectively. In Section 4.3, we review TCG techniques using model checking in literature.

4.1 Quantifier Elimination

Recall that a Hoare triple (see e.g. [51]) can be used as a requirements specification for a numerical program \( S \) under test:

\[
\{ \text{Pre} \} \ S \ \{ \text{Post} \},
\]

the precondition \( \text{Pre} \) and postcondition \( \text{Post} \) are first-order logic formulae over the reals (this approach uses the reals as an idealised model of all floating point data types). An SUT \( S \) with \( n \) inputs and 1 output can be modeled (and inferred) as
CHAPTER 4. CONSTRAINT SOLVING FOR TEST CASE GENERATION

a set of overlapping piecewise polynomials $p_0(\bar{x}), p_1(\bar{x}), \ldots, p_k(\bar{x})$, where $\bar{x}$ is a $n$-dimensional vector. This model can be easily generalised to $m$ outputs (c.f. Section 3.2).

To generate a test case for a numerical program specified and modelled as above, we solve a constraint defined on a local polynomial $p_i(\bar{x})$ as follows:

$$\mathbb{R} \models x_0^2 + x_1^2 + \cdots + x_{n-1}^2 \leq r^2 \land \text{Pre}(\bar{x}) \land \neg \text{Post}(\bar{x}) \land p_i(\bar{x}) = 0$$

Where $\bar{x} = x_0, x_1, \ldots, x_{n-1}$ and $\text{Pre}(\bar{x})$ and $\text{Post}(\bar{x})$ may be quantified by universal and/or existential quantifiers, $\forall x$ and $\exists x$. In this case, the underlying constraint solving algorithm has to solve the polynomial equalities and inequalities, and more importantly has to address the problem of quantifier elimination (QE).

A QE method for real closed field is an algorithm that accepts as input any first-order formula of reals and outputs an equivalent quantifier-free formula containing the same free variables (unqualified). We choose a powerful and efficiently implemented algorithm (in Mathematica), the Hoon-Collins cylindric algebraic decomposition (CAD) (see e.g. [19, 14]) to be the constraint solving algorithm for this specific problem setting. For more details about the application of CAD in LBT, see Paper I.

4.2 Model Checking Reactive Systems by Narrowing

In this section, we first give some background and preliminaries on model checking. Then we present our approach: model checking LTL formulas for extended Mealy automata and describe a reformulation of bounded model checking as a disunification problem that can be solved by narrowing.

4.2.1 Background on Model Checking

A Kripke structure is a type of state transition graph model used to capture the behaviour of reactive systems.

**Definition 4.2.1 (Kripke Structure)** Let $\text{AP}$ be a set of atomic propositions, a Kripke structure is a tuple $K = (S, S_0, R, L)$ consisting of:

- $S$, a finite set of states.
- $S_0$, a set of initial states, where $S_0 \subseteq S$.
- $R \subseteq S \times S$, a transition relation, where $\forall s \in S, \exists s' \in S$ such that $(s, s') \in R$.
- $L : S \rightarrow 2^{\text{AP}}$, a labelling function that assigns a set of atomic propositions to each state.

A path $\pi$ of a Kripke structure $K$ is an infinite sequence of states $\pi = s_0, s_1, s_2, \ldots$ such that $\forall i \geq 0, R(s_i, s_{i+1})$ holds. Since sometimes an infinite path is not usable in
4.2. MODEL CHECKING REACTIVE SYSTEMS BY NARROWING

practice, a trace is defined to refer to a finite sequence of states. A trace is usually a prefix of a path or a path that contains a transition loop.

Three common temporal logics are linear time logic LTL [70], Computation Tree Logic CTL [16], and the superset CTL* of these logics, which was introduced by Emerson and Halpern in [25]. Most existing model checkers support verifying both LTL and CTL formulas and here we only discuss LTL. In LTL, we consider modal logic operators such as $X$ for next time, $G$ for always (globally), $F$ for eventually (in the future), $U$ for until and $R$ for release.

An LTL formula $\phi$ can be satisfied by a path $\pi$ of model $M$, denoted by $M, \pi \models \phi$. The semantics of LTL is expressed for infinite paths of a Kripke structure. We let $\pi_i$ denote the suffix of the path $\pi$ starting from the $i$-th state, and $\pi_i$ denotes the $i$-th state of the path $\pi$, with $i \in \mathbb{N}$. The initial state of a path $\pi$ is $\pi_0$.

Definition 4.2.2 (LTL Semantics) The satisfaction relation for LTL formulas with respect to a path $\pi$ on a Kripke Structure $M$ is inductively defined as follows.

1. $M, \pi \models true$; $M, \pi \not\models false$
2. $M, \pi \models \phi$ iff $\phi \in L(\pi_0)$
3. $M, \pi \models \neg \phi$ iff $M, \pi \not\models \phi$
4. $M, \pi \models \phi \land \psi$ iff $M, \pi \models \phi$ and $M, \pi \models \psi$
5. $M, \pi \models \phi \lor \psi$ iff $M, \pi \models \phi$ or $M, \pi \models \psi$
6. $M, \pi \models \phi \rightarrow \psi$ iff $M, \pi \not\models \phi$ or $M, \pi \models \psi$
7. $M, \pi \models X\phi$ iff $M, \pi^1 \models \phi$
8. $M, \pi \models F\phi$ iff $M, \pi^i \models \phi$, for some $i \geq 1$
9. $M, \pi \models G\phi$ iff for all $i \geq 1$, $M, \pi^i \models \phi$

Based on a Kripke structure, a property is simply a set of infinite sequences of states, and properties are normally classified into two kinds: safety properties and liveness properties. Intuitively, a safety property expresses that something (bad) never happens while a liveness property states that something good eventually takes place. In formal verification, the distinction between safety properties and liveness properties allows us to choose the most appropriate proof technique with respect to these properties. In the context of testing, it is even more important to distinguish them since the violations of safety properties correspond to counterexamples of finite length, and the violations of liveness properties give counterexamples of infinite length. This means that violations of liveness properties may never be confirmed by testing or monitoring.

There are two main paradigms in earlier research on model checking: explicit model checking and symbolic model checking. In explicit model checking, the state space is explicitly represented and a search (usually depth-first search) is performed
through the model of a system one state at a time. To tackle the state explosion problem, symbolic model checking [53] is an important development, which uses ordered binary decision diagrams (OBDD) to represent states and their transition relations in a compact way. In [54] and [12] it is shown that symbolic model checking allows significantly larger state spaces than explicit state model checking.

Bounded model checking [9] is a recent development in model checking, which concerns finding counterexamples within bounded paths. Bounded model checking is reformulated into a propositional satisfiability (SAT) problem that can be solved by propositional decision procedures.

4.2.2 Principles of Abstract Data Types

Our model checking technique involves the theories of abstract data types and term rewriting, which are based on the notation and terminology of many-sorted algebra [55]. An $S$-sorted signature $\Sigma$ consists of an $S^* \times S$-indexed family of sets $\Sigma = \langle \Sigma_{\omega, s} | \omega \in S^*, s \in S \rangle$, where $c \in \Sigma_{e, s}$ is a constant symbol of sort $s$ if $s \in S$ and $f \in \Sigma_{\omega, s}$ is a function symbol if $\omega = s_1, \ldots, s_n \in S^+$.

We define $X = \langle X_s | s \in S \rangle$ an $S$-indexed family of disjoint sets $X_s$ of variables of sort $s$. In the sequel, the set $T(\Sigma, X)_s$ of all terms of sort $s \in S$ can be inductively constructed by:

- constant symbols of sort $s$: $c \in T(\Sigma, X)_s$ for $c \in \Sigma_{e, s}$
- variable symbols of sort $s$: $x \in T(\Sigma, X)_s$ for $x \in X_s$
- function symbols of codomain type $s$: $f(t_1, \ldots, t_n) \in T(\Sigma, X)_s$ for $f \in \Sigma_{\omega, s}$, $\omega = s_1, \ldots, s_n$ and $t_i \in T(\Sigma, X)_{s_i}$

The algebraic semantics of a system can be specified as an initial specification or model [50]. An initial specification is a pair $(\Sigma, E)$, where $E$ is an equation set defined over ADTs represented as a many-sorted $\Sigma$ algebra. It encapsulates data and functions (operations) that manipulate that data, and equational specifications (axioms). For example, the signature $\Sigma_{\text{Stack}}$ of the data type stack is given as follows:

- Sorts of $\Sigma_{\text{Stack}}$, $S = \{\text{element, stack}\}$
- Constant symbols of $\Sigma_{\text{Stack}}$ are:
  - $e_0, \ldots, e_n, \text{Error} : \rightarrow \text{element}$
  - $\text{Empty} : \rightarrow \text{stack}$
  - $\text{Push} : \text{element} \times \text{stack} \rightarrow \text{stack}$
  - $\text{Pop} : \text{stack} \rightarrow \text{stack}$
  - $\text{Top} : \text{stack} \rightarrow \text{element}$

and its equation set $E$ is:
4.2. MODEL CHECKING REACTIVE SYSTEMS BY NARROWING

- \( \text{Pop}(\text{Push}(x, s)) = s \)
- \( \text{Top}(\text{Push}(x, s)) = x \)
- \( \text{Pop}(\text{Empty}) = \text{Empty} \)
- \( \text{Top}(\text{Empty}) = \text{Error} \),

where \( x \in X_{\text{element}}, s \in X_{\text{stack}} \). Note that many simple systems have no axioms, e.g. \( E = \emptyset \) and \( \Sigma \) contains only constant symbols in the specification of TCP/IP stack in Paper II.

4.2.3 Extended Mealy Automata over Abstract Data Types

As discussed in Section 1.2.1, a reactive system is normally formalised as a Mealy automaton. To enhance the level of data expressiveness and abstraction, we define a Mealy automaton over abstract data types, termed an extended Mealy automaton (EMA). Moreover, we can model an EMA as a many-sorted algebraic structure by considering inputs, states and outputs as distinguished data types, and the input and output data types are common data types and structures such as \textit{int}, \textit{string}, \textit{array}, \textit{list} etc.

**Definition 4.2.3** A signature \( S = (S, \Sigma, \text{input}, \text{output}) \) for a Mealy automaton is a four-tuple, where \( S = \{\text{state}, s_1, \ldots, s_n\} \) a sort set, \( \Sigma \) is an \( S \)-sorted signature with distinguished constant and function symbols.

\[ q^0 \in \Sigma_{\text{state}}, \quad \delta \in \Sigma_{\text{state input}}, \quad \lambda \in \Sigma_{\text{state input}}, \quad \text{input}, \text{output} \in \{s_1, \ldots, s_n\} \]

are distinguished input and output types.

We say \( \Sigma \) is reachable if \( \Sigma \) has no other state constant symbols than \( q^0 \) and no other function symbols with codomain type \textit{state} than \( \delta \). \( A \) is a reachable EMA if \( \Sigma \) is reachable. We model an extended Mealy automaton \( A \) (of signature \( \Sigma \)) as an \( S \)-sorted \( \Sigma \) algebra \( A \), where \( q^0 \) is the initial state, \( \sigma_A : A_{\text{state}} \times A_{\text{input}} \rightarrow A_{\text{state}} \)
and \( \lambda_A : A_{\text{state}} \times A_{\text{input}} \rightarrow A_{\text{output}} \) are the state transition and output functions.

We define the extended state transition and output functions:

\[ \delta^*_A : A_{\text{state}} \times A_{\text{input}}^* \rightarrow A_{\text{state}}, \quad \lambda_A^* : A_{\text{state}} \times A_{\text{input}}^* \rightarrow A_{\text{output}} \]

which allows each reachable state of \( A \) to be represented as a finite sequence of input symbols starting from the initial state.

4.2.4 Linear Temporal Logic Specifications Over EMA

To express user requirements specifications over an EMA, we introduce an LTL language which integrates a data type signature of input and output values with propositional LTL. In the sequel we use two distinguished variable symbols, \( in \in X_{\text{input}} \) and \( out \in X_{\text{output}} \) for input and output data on transitions of an EMA.
Definition 4.2.4  The set LTL(Σ, X) of all linear temporal logic formulas over Σ and X is defined to be the smallest set of formulas containing the atomic proposition true and all equations (t = t') for each sort s ∈ S − {state} and all terms t, t ∈ T(Σ, X)s, which is closed under negation ¬, conjunction ∧, disjunction ∨, and the next X, always future G, sometime future F, always past G⁻¹, and sometime past F⁻¹ temporal operators.

Although past-time operators such as G⁻¹ and F⁻¹ do not add expressive power to pure future LTL according to [29], including such past-time operators makes this LTL exponentially more succinct as shown in [52].

Let φ denote the atomic propositions in the LTL language. As usual, X(φ) denotes that φ is true in the next input or time instant, F(φ) (and G(φ)) denotes that φ holds always (respectively at some time) in the future on a path or a run. In addition, we let φ → φ denote ¬φ ∨ φ, and t ≠ t' denote ¬(t = t'). For example, on an EMA A ∈ Alg(Σ), the LTL formula

\[ G((in = x) ∧ X((in = y) \implies X(out = x + y) )) \]

that at all times, if the current input is x and next input is y then in two time steps from now the output will be the sum x+y.

It is well known that for every formula φ ∈ LTL(Σ, X) there exists a logically equivalent formula φ' ∈ LTL(Σ, X) in negation normal form (NNF) where negations only occur in front of atomic subformulas. To model check an LTL formula on an EMA, we can translate its equivalent NNF LTL formula into a finite set of S = {S₁, ..., Sₙ} of constraint sets, where each constraint set Sᵢ consists of equations and negated equations. An equation (or negated equation) is a mathematical statement that asserts the equality (or inequality) of two terms. For t, t' ∈ T(Σ, X)s, an equation and negated equation are respectively a formula of the form (t = t') and t ≠ t'.

In this way an LTL formula is translated into a collection of constraint sets consisting of equations and negated equations. The model checking problem is then to search for a satisfying assignment to free variables in each set, which is known as disunification and can be solved by narrowing. This will be discussed in Section 4.2.5.

Let SatSet₁(φ) denote the collection of constraints sets that is the translation of any NNF formula φ. The SatSetᵢ(φ) can be inductively defined starting from the translation of equations and negated equations (atomic propositions) to formulas constructed by joining subformulas using logic and modal operators. The translation requires an additional set \( \bar{X} = \{ \bar{x}_i | x_i ∈ X_{input} \} \) of free variables representing input sequence elements in bounded steps. For equations and negated equations in φ, the variable symbol \( x_i \) occurring at step \( n \) is substituted by \( \bar{x}_n ∈ \bar{X} \), while the variable symbol \( out \) is substituted by the output transition function \( λ^* \) constructed from the initial state \( q_0 \), state transition function \( δ^* \) and input sequence \( \bar{X} \) leading to \( out \). For example, SatSet₂(in = x) = \{ \{ \bar{x}_2 = x \} \} and SatSet₂(out = y) = \{ \{ λ(δ^*(q_0, \bar{x}_0, \bar{x}_1), \bar{x}_2) = y \} \}. 
4.2. MODEL CHECKING REACTIVE SYSTEMS BY NARROWING

A complete and formal description of this translation is given in Definition 5 of Paper II, and the completeness and soundness of this translation is given in Theorem 1 of Paper II, which proves that it is necessary and sufficient to solve one of the constraint sets $S_1, \ldots, S_n \in \text{SatSet}_n(\phi)$ to solve an NNF formula $\phi$.

4.2.5 Narrowing for Solving Disunification Problems

Unification and Disunification

To discuss unification and disunification we need to introduce some technical concepts. A substitution $\sigma$ is an $S$-indexed family of mappings $\sigma_s : X_s \rightarrow T(\Sigma, X)_s$, such that $\{x \in X_s | \sigma_s(x) \neq x\}$ is finite for $s \in S$. This set is called the domain of $\sigma$ and denoted by $\text{Dom} \sigma$. A substitution $\sigma$ is usually identified as the set $\{x \mapsto \sigma(x) | x \in D\}$ and the empty substitution is $\emptyset$. The result of applying a substitution $\sigma$ to a term $t \in T(\Sigma, X)_s$, denoted by $\sigma(t)$ or $\sigma t$ is defined inductively:

(i) $\sigma(t) = t$ for $t \in \Sigma, s$,
(ii) $\sigma(t) = \sigma(x)$ for $x \in X_s$,
(iii) $\sigma(t) = f(\sigma(t_1), \ldots, \sigma(t_n))$ for $t = f(t_1, \ldots, t_n)$ and $t_i \in T(\Sigma, X)_s$.

The composition of two substitutions $\sigma, \sigma'$ is denoted by $\sigma \circ \sigma'$. A variable renaming is a family of bijective substitutions $\sigma_s : X_s \rightarrow X_s$. A substitution $\sigma$ is more general than a substitution $\tau$, denoted by $\sigma \preceq \tau$, if there exists a substitution $\delta$ such that $\delta \circ \sigma = \tau$.

Given an equational theory $E$, a finite set $\{t_1 = t_1', \ldots, t_n = t_n'\}$ of equations to be solved, $E$-unification (see e.g. [6]) is the process of finding a substitution $\sigma$ as a solution such that $\{\sigma t_1 = E \sigma t_1', \ldots, \sigma t_n = E \sigma t_n'\}$. Syntactic unification is a special case of unification for $E = \emptyset$, where the solution $\sigma$ makes $t_i$ and $t_i'$ syntactically identical.

Generally, in the case that a finite set $\{t_1 = t_1', \ldots, t_n = t_n', u_1 \neq u_1', \ldots, u_m \neq u_m'\}$ of equations and negated equations must be solved, this problem is known as the disunification problem (see e.g. [20]). Formally speaking, given a set of equations $E$, a disunification problem $S = \{(t_i, Q_i, t_i') | i = 1, \ldots, n, n \geq 0, Q_i \in \{=, \neq\}\}$ is a finite (possibly empty) set of equations and negated equations over a many-sorted data type signature $\Sigma$ and $X$. A substitution $\sigma : X \rightarrow T(\Sigma, X)$ is an $E$-unifier of a set $S$ if for all $1 \leq i \leq n$, if $Q_i$ is $=_{\Sigma}$ then $\sigma(t_i) =_{E} \sigma(t_i')$, and if $Q_i$ is $\neq_{\Sigma}$ then $\sigma(t_i) \neq_{E} \sigma(t_i')$. In particular, let $E$ be an equational data type specification having a complete rewrite system $R$, and $=_{R}$ be the reflexive symmetric and transitive closure of $R_S$. A substitution $\sigma : X \rightarrow T(\Sigma, X)$ is an $R$-unifier of $S$ if for all $1 \leq i \leq n$, if $Q_i$ is $=_{\Sigma}$ then $\sigma(t_i) =_{R} \sigma(t_i')$, and if $Q_i$ is $\neq_{\Sigma}$ then $\sigma(t_i) =_{R} \sigma(t_i')$.

We let $U(S)$ denote the set of all $E$-unifiers or $R$-unifiers of $S$. A unifier $\sigma \in U(S)$ is a most general unifier (mgu) if $\sigma \preceq \tau$ for all $\tau \in U(S)$. Similarly, syntactic disunification is a special case of disunification for $E = \emptyset$ ($R = \emptyset$), where $\sigma t_i$ and $\sigma t_i'$ syntactically identical if $Q_i$ is $=_{\Sigma}$ or different if $Q_i$ is $\neq_{\Sigma}$.

Syntactic unification and disunification problems are both decidable. Solving these problems can be expressed as an iterative transformation of a set of equations (and negated equations) using a set of rules (see [7, 20]), until the solution is trivial or no more transformations rules can be applied, which is termed solved form.
Definition 4.2.5 A unification problem \{x_1 = t_1, \ldots, x_n = t_n\} is in solved form if the \(x_i\) are pairwise distinct variables and none of \(x_i\) occurs in any of the \(t_i\). A disunification problem \{x_1 = t_1, \ldots, x_n = t_n, y_1 \neq t_1, \ldots, y_m \neq t_m\} is in solved form if the \(x_i\) are pairwise distinct variables, \(y_i \in \{x_1, \ldots, x_n\}\) and none of \(x_i\) (or \(y_i\)) occurs in any of \(t_i\) (or \(u_i\)).

In many important cases, however, both the unification and disunification problems are undecidable. Nevertheless, these problems are semidecidable and one can consider algorithms which are solution complete in the sense that they always terminate when a solution can be found. A large collection of such algorithms arises through the technique known as narrowing, which will be discussed in the remaining of this subsection.

Term Rewriting and Narrowing

We first introduce some notions of term rewriting before discussing narrowing. The notion of position in a term is used to describe the subterm occurrences. We let \(O(t)\) denote the set of all positions in \(t\) for \(t \in T(\Sigma, X)_s\), which is inductively defined by:

\[
O(t) = \begin{cases} 
\{\varepsilon\}, & \text{if } t = e, x \\
\{\varepsilon, k \cdot i \mid 1 \leq k \leq n, i \in O(t_k)\}, & \text{if } t = f(t_1, \ldots, t_n) 
\end{cases}
\]

The subterm at position \(p \in O(t)\) in \(t\) is denoted by \(t|_p\) and for \(t\) and \(u \in T(\Sigma, X)_s\), \(t|_p\) denotes the term obtained by replacing the subterm found at \(p\) in \(t\) by \(u\). We say that \(p \in O(t)\) is a non-variable position if \(t|_p\) is not a variable, and let \(\overline{O}(t)\) denote the set of all such non-variable positions.

A term rewriting rule is a directed equation of the form \(l \rightarrow r\) for \(l, r \in T(\Sigma, X)_s\) and \(s \in S\) such that \(\text{Var}(r) \subseteq \text{Var}(l)\), if \(\text{Vars}(t)\) denotes the set of all variables from \(X\) occurring in \(t\). A term rewriting system (TRS) \(R\) is a set of rewriting rules. If \(\sigma_e : X_e \rightarrow X_e\) is a family of variable renamings then \(\sigma(l) \rightarrow \sigma(r)\) is a variant of \(l \rightarrow r\). The (one step) rewrite relation \(\xrightarrow{R}\) associated with a TRS \(R\) is a binary relation on terms defined by \(t \xrightarrow{R} t'\) if there exists a rule \(l \rightarrow r\), a position \(p \in O(t)\) and a substitution \(\sigma\) such that \(t|_p \equiv \sigma(l)\) and \(t' \equiv t[\sigma(r)]|_p\). We let \(t \xrightarrow{R^*}\) denote the reflexive transitive closure (many step rewriting relation) of \(t \xrightarrow{R}\).

A TRS \(R\) is strongly normalising if there is no infinite sequence of terms \(t_0, t_1, \ldots\) such that \(t_0 \xrightarrow{R} t_1 \xrightarrow{R} t_2 \xrightarrow{R} \ldots\) and \(R\) is confluent if for any term \(t, t_1, t_2 \in T(\Sigma, X)_s\), if \(t \xrightarrow{R} t_1\) and \(t \xrightarrow{R} t_2\) then there exists \(t' \in T(\Sigma, X)_s\) such that \(t_1 \xrightarrow{R} t'\) and \(t_2 \xrightarrow{R} t'\). A complete TRS is confluent and strongly normalising.

Narrowing is initially studied in the context of \(E\)-unification. It is shown in [46] that the method of narrowing in term rewriting gives a complete search algorithm whenever the theory \(E\) can be represented by a complete TRS \(R\). Informally speaking, narrowing a term is applying to it the (minimum) substitution (normally
We say that a term \( t \) is \( R \)-narrowable into a term \( t' \) if there exists a non-variable position \( p \in \mathbb{O}(t) \), a variant \( l \rightarrow r \) of a rewrite rule in \( R \) and a substitution \( \sigma \) such that:

- \( \sigma \) is a most general syntactic unifier of \( t_p \) and \( l \), and

- \( t \equiv \sigma(t[r]^p) \).

We write \( t \leadsto_{[p,l\rightarrow r,\sigma]} t' \) or simply \( t \leadsto_{\sigma} t' \). The relation is called \( R \)-narrowing.

The \( R \)-narrowing relation on terms can be easily extended to equations and negated equations for solving a disunification problem \( S = \{(t_i, Q_i, t'_i) \mid i = 1, \ldots, n, n \geq 0, Q_i \in \{-, \neq}\}\). A formula \( (t Q t') \) is \( R \)-narrowable into a formula \( (u Q u') \) if there exists a variant \( l \rightarrow r \) of a rewrite rule in \( R \) and a substitution \( \sigma \) such that either \( t \leadsto_{[p,l\rightarrow r,\sigma]} u \) for some non-variable occurrence \( p \in \bar{O}(t) \) or \( t' \leadsto_{[q,l\rightarrow r,\sigma]} u' \) for some non-variable occurrence \( q \in \bar{O}(t') \). We write \( (t Q t') \leadsto_{[q,l\rightarrow r,\sigma]} (u Q u') \) or simply \( (t Q t) \leadsto_{\sigma} (u Q u') \) for the \( R \)-narrowing relation on equations or negated equations, and \( S \leadsto_{[q,l\rightarrow r,\sigma]} S' \) or \( S \leadsto_{\sigma} S' \) for the \( R \)-narrowing relation a disunification problem \( S \).

Narrowing is essentially an iterative process of searching for a terminated narrowing derivation \( S \leadsto_{\sigma_1} S_1 \leadsto_{\sigma_2} \ldots \leadsto_{\sigma_n} S_n \) such that \( S_n \) is syntactically unifiable by a substitution \( \theta \), i.e., \( \theta \circ \sigma_n \circ \cdots \circ \sigma_1 \) is an \( R \)-unifier of \( S \). As the search space of narrowing is large and narrowing procedures frequently fail to terminate, many proposals have been made to increase the efficiency of narrowing.

One important restriction on the set of occurrences available for narrowing, termed basic narrowing, was introduced by Hullot in [46], and has since been widely studied, e.g. [59]. A basic narrowing derivation is very similar to a narrowing derivation as given in Definition 4.2.6 above. However, in a basic narrowing derivation, narrowing is never applied to a subterm introduced by a previous narrowing substitution, which is called a non-basic position. In other words, the narrowing can only be applied to basic positions.

**Definition 4.2.7** Let \( t_1 \leadsto_{[p_1, l_1 \rightarrow r_1, \sigma_1]} t_2 \leadsto_{[p_2, l_2 \rightarrow r_2, \sigma_2]} \cdots \leadsto_{[p_{n-1}, l_{n-1} \rightarrow r_{n-1}, \delta_{n-1}]} t_n \) be a narrowing derivation. Basic positions \( B_1, \ldots, B_n \) can be inductively defined as follows:

\[
B_1 = \bar{O}(t_1),
\]

\[
B_{i+1} = B_i \cup \{q \in B_i \mid p_i \leq q\} \cup \{p_i \cdot q \mid q \in \bar{O}(r_i)\}, \quad i \geq 1.
\]

and positions in \( \bar{O}(t_i) - B_i \) are called non-basic positions.
For example, consider two TRSs $R$ and $R'$

\[
R = \begin{cases}
g(x) & \rightarrow f(x, x) \\
a & \rightarrow b \\
f(a, b) & \rightarrow c \\
f(b, b) & \rightarrow c,
\end{cases} \\
R' = \begin{cases}
g(x) & \rightarrow f(x, x) \\
a & \rightarrow b \\
f(a, b) & \rightarrow c \\
f(b, b) & \rightarrow g(a)
\end{cases}
\]

where $R$ is complete (confluent and strongly normalising), while $R'$ is not strongly normalising because it has infinite rewriting sequences, one of which is $g(a) \rightarrow f(a, a) \rightarrow f(a, b) \rightarrow f(b, b) \rightarrow g(a) \rightarrow \ldots$

To solve the problem $g(a) =_? c$ using $R$ by basic narrowing, a possible basic narrowing derivation is:

\[
g(a) \leadsto [g(x) \rightarrow f(x, x), x \mapsto a] f(a, b) \leadsto c =_? c.
\]

However, it is impossible to solve this problem using $R'$ by basic narrowing, since the only $R$-narrowing derivation that solves this problem is:

\[
g(a) \leadsto [g(x) \rightarrow f(x, x), x \mapsto a] f(a, b) \leadsto c =_? c,
\]

and this is not a basic $R'$-narrowing derivation. In this narrowing derivation, all the non-basic positions are marked by underlining and narrowing is applied to a non-basic position in the second step. From these two examples $R$ and $R'$, we see that the structure of a TRS is important for basic narrowing to succeed.

To yield a decision procedure for LTL model checking, basic $R$-narrowing satisfies the following properties:

- Sound and complete. Basic $R$-narrowing is sound and complete in the sense of Theorem 2 of Paper II.
- Terminating. This property is important in the context of TCG, where we need to know if a test case exists at all. As shown in [46] (also Theorem 3 of Paper II), every $R$-narrowing derivation terminates, if any basic $R$-narrowing derivation from any of right hand sides of rules in $R$ terminates. According to [57], an EMA can be learned as a complete TRS $R_{ema}$ which contains no free variables (rewriting from ground terms to ground terms). This means that it is impossible to construct an $R_{ema}$-narrowing derivation from any the right hand sides of rules in $R_{ema}$, which is an special instance of Theorem 3 of Paper II. Then every $R_{ema}$-narrowing derivation terminates.

### 4.3 Related Work

As suggested in Section 2.3, both LBT and MBT use model checking techniques for TCG, but in a substantially different way due to the different test objectives.
4.3. RELATED WORK

In LBT, the test objective refers to user requirements formalised as LTL formulas. In MBT, however, test objectives are derived according to a coverage criterion [43]. Most approaches use a model checker to check against the *never-claims* (negated test objectives) to produce counterexamples as test cases, but it is possible to use witness traces when model checkers are able to construct witnesses. Using model checking to automate TCG based on coverage satisfaction requires formalised test objectives, termed *test predicates*, which are coverage criteria expressed in temporal logic formulas.

When it comes to specifications involving Boolean expressions, e.g. transition predicates in EFSMs, a number of logic expression coverage criteria can be defined according to the hierarchy such as predicate coverage, (complete) clause coverage, general active clause coverage, modified condition/decision coverage (MC/DC) and so on (see [3, 27, 15]). Also, in [44, 42, 13] many coverage criteria based on control-flow and data-flow graphs, and transition systems can be specified as sets of temporal logic formulas, including *all state, all transition coverage, criteria based on definition-use pairs* and so on. Ammann et al. proposes in [4] to combine mutation analysis with model checking techniques by mutating a model and generating test cases distinguishing between these two models. For more coverage criteria and evaluation of all coverage criteria (including combinatorial coverage and user requirement coverage), see [27].
Chapter 5

Empirical Performance Evaluation

5.1 Methodology

It seems difficult to make a theoretical analysis of the performance of LBT algorithms. Therefore, we have considered the fault detection capability of LBT using empirical performance evaluation. We chose random testing as the most important benchmark to evaluate how much testing can be improved by using inductive learning and constraint solving techniques, which are absent in random testing.

As the performance of LBT is non-deterministic (due to the presence of random test cases) as is random testing, we compared their converged performance results averaged over a large number of runs for the same injected fault(s). As discussed in Section 2.2.1, it was easy to generate a large number of tests in random testing.

We considered two levels of performance: logical and real-time performance by measuring two related parameters needed to first discover an error injected into an SUT: (i) the number of test cases, denoted by $T_{first}$, and (ii) the time (in seconds), denoted by $t_{first}$. In logical performance evaluation, we compared $T_{first}$ for LBT and $T_{first}$ for random testing, while in real-time performance evaluation, we take time complexity of learning and constraint solving into consideration and compared the $t_{first}$ for LBT with $t_{first}$ for random testing.

In performance evaluation for numerical procedural systems in Paper I, we used executable models (randomly generated polynomials) and a sorting program as SUTs, which were mutated to contain errors. We also studied the error size and the logical complexity of specifications as influences on the results. A performance evaluation for reactive systems was given in Paper II, where the SUT was chosen to be a mutated TCP/IP stack model.

5.2 Evaluation for Procedural Systems

Using an executable model as an SUT can be considered as an abstract case study for performance evaluation of procedural systems. A procedural system can be
considered as a function with possibly multi-dimension input. To inject errors into such an SUT, we changed the function of the SUT partially in order to make the mutated part of the function falsify the postcondition in the specification.

We also considered a more concrete case study using the well known bubble sort algorithm. Since the input (and hence output for in-place sorting) array size is usually rather large, this algorithm represents a typical high dimensional problem, which was solved by using LBT combined with pair-wise testing (c.f. Section 2.2.2).

5.2.1 Random Generation of Numerical System Under Test

Instead of being limited to a small number of specific numerical SUTs, we used a random generator to construct the numerical SUTs, their specifications and mutations in a controlled way. This allows us to obtain the largest possible data set of performance results.

The random generator consists of the random numerical program generator, the random specification generator and the random mutation generator.

To randomly generate a numerical SUT, the random numerical program generator (RPG) divides a global \( n \)-dimensional input space into subspaces at random. Within each of these randomly chosen subspaces, the RPG generates an \( n \)-dimensional polynomial randomly (i.e. random coefficients and degree). We allowed for much higher degrees in such SUT models than the learned model in Section 3.2 in order to ensure that the learning problem is non-trivial.

For an SUT \( S \) generated by the RPG, random specification generator (SG) randomly generates a specification \( \text{Req} \) satisfied by the SUT \( S \). The random mutation generator (MG) randomly mutates \( S \) by changing/regenerating polynomial functions over some of the subspaces in such a way that the mutated SUT \( S' \) does not satisfy \( \text{Req} \).

To study the relative performance of LBT and random testing, we considered two important factors as follows:

- **Percentage error size.** Given an SUT generated by the RPG and mutated by the MG, the percentage error size is represented as the ratio of the total size of the mutated subspaces to the total size of all subspaces. Controlling this percentage error size experimentally, was the key to understanding the relative performance of LBT and random testing.

- **Logical complexity of requirements specification.** It is shown in [14] that the logical complexity of requirements formulas has an effect on the efficiency of constraint solving procedure of LBT. In order to explore the effects of this complexity, we studied the relative performance of LBT and random testing against two different logical types of requirements specifications, of which the precondition is always a lower and/or upper bound on the input space and the postcondition can be either equational or inequational.
Let $S$ be a randomly generated SUT with $k$ subspaces and $n$-dimensional input $S = f_1(\bar{x}), f_2(\bar{x}), \ldots, f_k(\bar{x})$, where $\bar{x}$ is a $n$ dimensional vector of input variables. The mutated SUT of $S$ is denoted by $S' = m_1(\bar{x}), m_2(\bar{x}), \ldots, m_k(\bar{x})$, where $m_i(\bar{x}) \equiv f_i(\bar{x})$ if subspace $i$ is not mutated, otherwise $m_i(\bar{x}) \neq f_i(\bar{x})$. Intuitively, the equational postcondition asserts that the mutated SUT $S'$ is equal to the expected SUT $S$, to within an absolute tolerance $\epsilon$, denoted by $\|m_i(\bar{x}) \equiv f_i(\bar{x})\| < \epsilon$ for $1 \leq i \leq k$. However, an inequational postcondition is less strict. It asserts that all values of each polynomial function $f_i(\bar{x})$ for $1 \leq i \leq k$ lie above a constant lower bound (or below a constant upper bound). More details about random generation of numerical SUTs can be found in Paper I.

5.2.2 Results and Analysis

For the abstract case study, we generated a large number of SUT/specification pairs, and mutated these SUTs to achieve different percentage error sizes within the range 10% to 0.01% using the random generator. We then measured and compared the value of $T_{\text{first}}$ (c.f. Section 5.1) using LBT and random testing over many different SUTs and different mutations all of the same percentage error size. Moreover, the results clearly indicated that the relative performance of LBT was influenced by the percentage error size and the logical complexity of specifications. The results of our experiments illustrated that the efficiency of LBT over random testing increases as the percentage error size decreases for both equational and inequational postconditions. On the other hand, the relative performance of LBT is also influenced by the logical complexity of specification: LBT is more efficient than RT for inequational postconditions than for equational postconditions.

In the concrete case study, we introduce a mutation into the bubble sort algorithm source code and our results showed that on average LBT is 10 times faster than random testing at uncovering this mutation error.

5.3 Evaluation for Reactive Systems

We compare LBT with random testing on the TCP/IP stack model (a Mealy automaton) by measuring $T_{\text{first}}$ and $t_{\text{first}}$ over 1000 system runs using the same injected errors. The results of testing five different requirements for TCP/IP shows that LBT always finds errors with significantly fewer test cases ranging between 0.25% and 18% of the number required by random testing ($T_{\text{first}}$), and is often but not always significantly faster than random testing, ranging between 0.9% and 160% of the time required by random testing ($t_{\text{first}}$). By comparing the state size of the learned model and the SUT, we see that LBT can always find errors before the SUT is completely learned. This confirms the advantage of using an incremental learning algorithm such as CGE. It should also be pointed out that as real-time measurement involves factors such as efficiency of implementation, there also exists scope for further improvement of LBT on the implementation level.
Chapter 6

Conclusion and Future Work

6.1 Conclusion and Contribution

In this thesis, we have introduced LBT, a specification-based black-box testing approach using inductive learning, formal methods and constraint solving techniques. Given a formal requirements specification $\text{Req}$ of an SUT $S$, an LBT framework is essentially a (finite) iterative process that consists of an inductive learning algorithm for constructing the behavioural model $M$ of $S$ and a decision procedure based on constraint solving for TCG. At the $i$-th iteration of testing, LBT constructs a behavioural model $M_i$ of $S$ using executed test cases $T_i$ as observations, and searches for a counterexample of $\text{Req}$ on $M_i$ as a test case $t_i$ (randomly generated if the searching terminated with no counterexample). Then $t_i$ is executed on $S$ and the observed I/O behaviour of $S$ is evaluated against $\text{Req}$ as confirmation of the system fault. In the case where no fault is exhibited by the execution of $t_i$, LBT updates $T_i$ to $T_{i+1}$ by adding $t_i$ and continues with the next iteration.

We investigated testing two types of SUT models: procedural and reactive systems. For each type of system, we suggested a suitable I/O behavioural model, a requirement specification language, model inference and constraint solving algorithms, which are combined together to automate test case generation and evaluation. Since it is expensive or even infeasible to completely learn a complex SUT, we used incremental learning algorithms that are more efficient and appropriate in the context of testing.

To evaluate the performance of LBT, we were mainly concerned about fault detection capability, and hence we compared LBT against random testing as the benchmark with regards to the speed of uncovering injected errors. Our statistical results have shown the consistent efficiency of LBT over random testing and the potential scope to improve real-time performance.

The main contributions of this thesis are as follows:

- Two specification-based black-box testing frameworks (LBT) for procedural and reactive systems were investigated and both were evaluated against ran-
A suitable behaviour model for numerical programs, the piecewise polynomial model was introduced and an efficient learning algorithm was developed.

A suitable behaviour model for reactive systems, a compact representation of an EMA as a complete TRS was introduced, which can be incrementally and efficiently learned.

A bounded model checking algorithm by means of narrowing was proposed to automate TCG for reactive systems.

Both LBT frameworks were implemented and evaluated using case studies. In the case study for procedural systems, a framework for automated case study generation (the random SUT generator) was developed.

6.1.1 Author Contributions

The idea of investigating learning-based approaches to testing software and systems is due to my supervisor Karl Meinke. Based on the initial work [56], we jointly developed two LBT frameworks for procedural and reactive systems. I assisted with developing the learning algorithm for numerical programs in Paper I and the model checking algorithm in Paper II. I also implemented the systems and made the experiments for two LBT frameworks. I wrote parts of the papers published as the results of our joint work.

6.2 Future Work

In future research, we will be mainly concerned about testing reactive systems. Work on improving automata learning and model checking algorithms will be considered such as the following:

- *Local learning algorithms for EMA*. Recall that we model a procedural system as piecewise polynomials, which are locally and incrementally inferred. We can also measure convergence of local models and generate test cases over just the best converged local models. This substantially increases the efficiency of constraint solving and improves the quality of generated test cases. To improve the efficiency of learning and model checking for reactive systems, we can investigate local learning and model checking for EMA in the following ways:

  - Define the *distance* value between two states in an EMA. As states are represented as input strings starting from the initial state, computing this distance value can be inspired by well-known *edit distances* of two strings such as Hamming distance, Levenshtein distance and so on.
When merging an equivalent class of states, these states must be close to each other, i.e. within a certain distance. This restriction on state merging will substantially decrease the complexity of learning (e.g. the number of possible rewriting rules in CGE [57]) and it would lead to a much faster EMA learning algorithm (possibly linear-time complexity) than CGE (quadratic-time complexity).

One issue that local learning may bring in is that it doesn’t guarantee that learned models are minimal. This means more efficient model checking algorithms are needed to deal with the extra complexity added by the learning algorithm.

- **An active CGE algorithm.** In the case where the model checker doesn’t return any counterexample at all, LBT proceeds with generating a test case in an alternative approach, which is currently by random test case generation. Another possible approach could be to adapt CGE to be active and thus the learning algorithm can request certain queries (test cases) to be executed when no test cases can be generated from model checking. The active CGE algorithm may converge relatively faster than the original CGE as it can gather essential information from the teacher.

- **Learning infinite-state EMA.** An EMA can have infinite states, in which input and output values contain free variables over infinite data types such as `int`, `string` and so on. Therefore, it would be interesting to adapt CGE for learning infinite-state EMA, which would possibly involve abstraction techniques (see e.g. [1]).

- **Combining narrowing and theorem proving.** An important observation in the experiments of Paper II is that counterexamples to correctness may be sparse, in which case narrowing fails very slowly. This suggests that a pure narrowing procedure could be significantly improved by interleaving a process of detecting unsatisfiability, e.g. theorem proving techniques.

- **More narrowing strategies.** It is known that basic narrowing modulo theories is incomplete for a unification problem $\exists x t = E t'$, where $E$ contains a set of axioms, in particular associativity-commutativity (AC). Recent development such as variant narrowing [26] could be considered in our LBT framework to solve such problems.

- **More case studies.** The SUTs we used for performance evaluations in Papers I and II are fairly small in size. We will have to consider more industrial-strength case studies to further evaluate and improve the efficiency and scalability of our LBT framework.
Bibliography


A Learning-based Approach to Unit Testing of Numerical Software

Karl Meinke and Fei Niu
Royal Institute of Technology, Stockholm
karlm@csc.kth.se niu@csc.kth.se

Abstract. We present an application of learning-based testing to the problem of automated test case generation (ATCG) for numerical software. Our approach uses n-dimensional polynomial models as an algorithmically learned abstraction of the SUT which supports n-wise testing. Test cases are iteratively generated by applying a satisfiability algorithm to first-order program specifications over real closed fields and iteratively refined piecewise polynomial models.

We benchmark the performance of our iterative ATCG algorithm against iterative random testing, and empirically analyse its performance in finding injected errors in numerical codes. Our results show that for software with small errors, or long mean time to failure, learning-based testing is increasingly more efficient than iterative random testing.

1 Introduction

For black-box specification-based testing, (see e.g. [11]) an important scientific goal is automated test case generation (ATCG) from a formal requirements specification, by means of an efficient and practical algorithm. A general approach common to several tools is to apply a satisfiability algorithm to a formal specification and/or a code model in order to generate counterexamples (test cases) to correctness. For an SUT that involves floating point computations one important problem is therefore to find an an expressive formal requirements language suitable for modeling floating point requirements together with an efficient satisfiability algorithm for generating floating point counterexamples. One possibility is to consider the first-order language and theory of real closed fields for which satisfiability algorithms have been known since [14].

To achieve high coverage levels it is important to go beyond individual test case generation towards iterative testing techniques that can iteratively generate a large number of high quality test cases in a reasonable time. In [10] we identified one such iterative heuristic for ATCG that we term learning-based testing (LBT). Our earlier work concerned black-box unit testing of numerical programs based on:

(i) simple learning algorithms for 1-dimensional problems,
(ii) simple requirements specifications which are quantifier free first-order formulas, and
(iii) an elementary satisfiability algorithm based on algebraic root solving for cubic 1-dimensional polynomials.

In this paper we present a systematic and more powerful extension of LBT that is suitable for black-box testing of complex numerical software units, including high-dimensional problems by means of n-wise (e.g. pairwise) testing. We apply n-dimensional polynomial learned models that can support n-wise testing and thus tackle the dimensionality problem associated with software units. We generalise our previous learning algorithm to support n-dimensional piecewise polynomial models using non-gridded data. This step also tackles the dimensionality problem on the SUT level. Finally, we use the Hoon-Collins cylindric algebraic decomposition (CAD) algorithm for satisfiability testing (see e.g. [1]). This is a powerful satisfiability algorithm for first order formulas over the language of real closed fields. Thus we greatly expand the complexity of the requirements specifications that our tool can deal with.

It is natural to question the achieved quality of the test cases generated by any new TCG method. In the absence of a theoretical model of efficiency, we have benchmarked the quality of our learning-based ATCG empirically, by comparing its performance with that of an iterative random test case generator. This was the only other iterative ATCG method for floating point computations to which we had any access. Since iterative random testing (IRT) is closely related to measures of mean time to failure (MTF), our benchmarking results have a natural and intuitive interpretation.

To carry out this performance comparison systematically, it was necessary to avoid the experimental bias of focussing on just a small number of specific SUTs and specific requirements specifications. To generate the largest possible amount of comparative data we automated the synthesis of a large number of numerical SUTs, their black-box specifications and their mutations. In this way we compared the performance of learning-based and iterative random testing over tens of thousands of case studies. Our average case results over this data set demonstrate that when mutated errors have small effects (or equivalently when the MTF of the mutated SUT is long) then learning based testing is increasingly superior to random testing.

The structure of this paper is as follows. In Section 1.1 we review some related approaches to unit testing of numerical software and ATCG, found in the literature. In Section 2, we discuss the general principles of learning-based testing. In Section 3, we present our learning-based testing algorithm for numerical programs. In Section 4, we describe our evaluation technique, and present the results of evaluation. In Section 5 we draw some conclusions from our work.

1.1 Related Work

There are two fields of scientific research that are closely related to our own. On the one hand there is a small and older body of research on TCG for scientific computations. On the other hand, there is a more recent growing body of research on ATCG methods obtained by combining satisfiability algorithms
with computational learning. Finally, there is an extensive mathematical theory of polynomial approximation, see for example [12].

**Unit Testing for Scientific Software** It is widely recognised that the problem of testing scientific software has received relatively little attention in the literature on testing. The existing literature seems focussed either on manual techniques for TCG, or ATCG for individual test cases. Besides iterative random testing, we are not aware of any other approach to iterative ATCG for numerical programs.

The intrinsic unreliability of many existing libraries of scientific code for the earth sciences has been empirically studied at length in [6] and [7], which cite static error rates in commercial code of between 0.6% and 20% depending upon the type of program error.

The correctness of scientific library codes for metrology is considered in [4]. Here the authors consider black-box unit testing of well specified computations such as arithmetic mean and deviation, straight-line and polynomial regression. They apply 1-wise testing where all fixed parameter values and one variable parameter value are chosen from sets of reference data points that are manually constructed to span a given input profile. Reference values are then algorithmically computed using numerical codes which the authors posit as providing reliable benchmarks. The actual outputs produced by the SUT are compared with these reference values using a composite performance measure. This measure is more sophisticated than an individual error bound as it can account for the degree of difficulty of the chosen reference data set. The approach seems more oriented towards measuring overall code quality while our own approach focusses on finding individual coding errors quickly. Nevertheless, [4] successfully highlights problems of numerical stability occurring in a variety of well known scientific libraries and packages including NAG, IMSL, Microsoft Excel, LabVIEW and Matlab, thus confirming the real problem of correctness even among simple and widely used scientific codes.

The method of manufactured solutions (MMS) presented in [13] provides a theoretically well-founded and rigorous approach for generating a finite set of test cases with which to test a numerical solver for a PDE (usually non-linear). The test cases are constructed by analysis of the underlying mathematical model as a non-linear system operator \( L[u(x, y, z, t)] \) to produce a finite set of analytical solutions for different input parameter values. Like our own approach (but unlike [4]) MMS does not make use of any benchmark or reference codes for producing test cases. This method is shown to be effective at discovering order-of-accuracy errors in fault injected codes in [8]. The main weakness of MMS is its restriction to a specific class of numerical programs (PDE solvers). By contrast, the generality of our requirements language (first-order logic) and our modeling technique (piecewise polynomials) means that by general results such as the Weierstrass Approximation Theorem, we can perform ATCG for a much larger class of programs and requirements.

3
ATCG using Computational Learning  Within the field of verification and testing for reactive systems, the existence of efficient satisfiability algorithms for automata and temporal logic (known as model checkers) has made it feasible to apply computational learning to ATCG using similar principles to our own. This approach is known as counterexample guided abstraction refinement (CEGAR). Some important contributions include [3], [2] and [5]. The basic principles of CEGAR are similar to those we outline in Section 2, though our own research emphasizes incremental learning algorithms and related convergence measures on models both to guide model construction and to yield abstract black-box coverage measures. While existing research on CEGAR emphasizes reactive systems, temporal logic and discrete data types, our work presented here considers procedural systems, first order logic and continuous (floating point) data types. However, the success of CEGAR research strongly suggests to generalize this approach to other models of computation, and our work can be seen as such a generalization.

2 Learning-based Testing

The paradigm for ATCG that we term learning-based testing is based on 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 heuristic iterative methods to search for and automatically generate a sequence of test cases until either an SUT error is found or a decision is made to terminate testing. The heuristic approach is based on learning a black-box system using tests as queries.

A learning-based testing algorithm should iterate the following five 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 \( (i_1, o_1), \ldots, (i_n, o_n) \) can be synthesized into a learned model \( M_n \) of \( S \) using an incremental learning algorithm. Importantly, this synthesis step involves a process of generalization from the data, which usually represents an incomplete description of \( S \). Generalization gives the possibility to predict as yet unseen errors within \( S \) during Step 2.

(Step 2) The system requirements \( \text{Req} \) are satisfiability checked against the learned model \( M_n \) derived in Step 1. This process searches for counterexamples to the requirements.

(Step 3) If several counterexamples are found in Step 2 then the most suitable candidate is chosen as the next test case input \( i_{n+1} \). One can for example attempt to identify the most credible candidate using theories of model convergence (see Section 3.3) or approximate learning.
(Step 4) The test case $i_{n+1}$ is executed 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 5. 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 synthesize a refined model $M_{n+1}$ of $S$.

(Step 5) We terminate with a true negative test case $(i_{n+1}, o_{n+1})$ for $S$.

Thus an LBT algorithm iterates Steps 1...4 until an SUT error is found (Step 5) 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. From the point of view of abstract black-box coverage, an interesting termination criterion is to choose a minimum value for the convergence degree $d_n$ of the model $M_n$ as measured by the difference $|M_n| - |M_{n-1}|$, according to some norm $|.|$ on models. We can then terminate when $M_n$ achieves this convergence degree.

This iterative approach to TCG yields a sequence of increasingly accurate models $M_0, M_1, M_2, \ldots, M_n$. (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. Notice 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.

3 Algorithm Description

In this section we show how the general framework of learning-based testing, described in Section 2, can be instantiated by: (i) piecewise polynomial models $M_i$, (ii) incremental learning algorithms, and (iii) an implementation of the CAD algorithm in Mathematica$^\text{TM}$, used as a satisfiability checker. The resulting learning-based ATCG can be used to automatically unit test numerical programs against their requirements specifications expressed as first order formulas over the language of real closed fields.

3.1 Piecewise Polynomial Models

The learned models $M_i$ that we use consist of a set of overlapping local models, where each local model is an $n$-dimensional and $d$-degree polynomial defined over an $n$-dimensional sphere of radius $r$ over the input/output space. Since $(d+1)^n$ points suffice to uniquely determine an $n$-dimensional degree $d$ polynomial, each local model has $(d+1)^n$ such points. One such point $c \in \mathbb{R}^n$ is distinguished as the centre point. The radius $r$ of a local model is the maximum of the Euclidean distances from the centre point to each other point in that model. Figure 1 illustrates this for the simple case $n = 1$ and two overlapping 1-dimensional
cubic polynomial models of degree 3. With $n$-dimensional polynomials we can automate $n$-wise testing, e.g. for $n = 2$ we obtain pairwise testing. Concretely, a model $M_i$ is represented as an array of $LocalModel$ objects.

![Diagram of overlapping areas and center points]

**Fig. 1.** Two cubic local models $f_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i$, $i = 1, 2$

We use a *non-gridded approach* to selecting the centre point for each local model. This means that the centre point $c$ can take any value $x \in \mathbb{R}^n$ and is not constrained to lie on a vertex of an $n$-dimensional grid of any specific mesh size. Non-gridded modeling helps avoid an exponential blowup in the number of grid vertex points as the model dimension $n$ increases. It also allows us to choose test case values with complete freedom. So test cases can cluster densely within areas of suspected errors, and sparsely within areas of likely correct behavior. This is one reason why LBT exceeds the performance of iterative random testing since the latter uniformly samples the search space of test cases.

### 3.2 A Learning-based Testing Algorithm

We now give a concrete instantiation of the abstract learning-based ATCG, outlined in Section 2, for numerical programs. This combines incremental learning methods for the piecewise polynomial models described in Section 3.1 together with an implementation of the CAD algorithm for satisfiability checking such polynomial models against requirements specifications. As is usual for procedural programs, a requirements specification for a numerical program $S$ under test is expressed as a Hoare triple (see e.g. [9])

$$\{pre\} S \{post\},$$

where the *precondition* $pre$ and *postcondition* $post$ are first order formulas over the language of real closed fields. Thus $pre$ and $post$ describe constraints on the input and output floating point variables of $S$ at the start and end of computation. A true negative (or failed) test case is any test case which satisfies the triple $\{pre\} S \{\neg post\}$ under the usual partial correctness interpretation of Hoare
triples. Our algorithm automatically searches for at least one true negative test case where \( S \) terminates. (We handle non-termination, should it arise, with a simple time-out.) This approach is easily extended to multiple negatives if these are required.

Before satisfiability checking can be applied, it is necessary to have at least one local polynomial model. Therefore the ATCG procedure given in Algorithm 1 below is divided into an initialisation phase and a learning-based phase. During the initialisation phase (lines 1-10), the minimum number \((d + 1)^n\) of test cases necessary to build one local model, is randomly generated (line 4). Each such test case \( t \) is executed on the SUT \( S \) and the output is stored (line 6). During the iterative learning-based phase (lines 11-30), on each iteration we try to generate a new test case either through: (i) performing satisfiability checking on the learned model (line 16) or, (ii) random test case generation (line 22). Whenever a new test case is generated and executed, the result is added to the model (line 26). Then the local models nearest the test case are updated and refined using this test case (line 28).

Note that Algorithm 1 makes two API calls to the Mathematica kernel\(^1\). Line 16 of Algorithm 1 makes a call to the Mathematica kernel in order to satisfiability check the formula \( pre \land \neg post \) against the local model \( m \), for each of the \( C \) best converged local models. The kernel function \( \text{FindInstance}(\ldots) \) is the Mathematica implementation of the Hoon-Collins CAD algorithm. If a satisfying variable assignment to \( pre \land \neg post \) over \( m \) exists then this kernel call returns such a variable assignment. In the case that no counterexample is found among the \( C \) best converged local models, then line 22 of Algorithm 1 makes a call \( \text{FindInstance}(M[\text{random()}], pre) \) to the same kernel function to find a satisfying variable assignment over a randomly chosen local model for precondition \( pre \).

As shown in Algorithm 1, an array \( M \) of \( \text{LocalModel} \) objects is maintained. We use \( M[i] \) for \( 0 \leq i < \text{length}(M) \) to denote the \( i \)-th element of \( M \) and \( M[i : j] \) for \( 0 \leq i \leq j < \text{length}(M) \) to refer to the subinterval of \( M \) between \( i \) and \( j - 1 \), inclusive.

### 3.3 Learning Local Models

The two subprocedures \( \text{LearnModel} \) and \( \text{UpdateModels} \) called in Algorithm 1 are detailed in Algorithms 2 and 3. These implement a simple incremental learning method along the general lines described in Step 1 of Section 2. Algorithm \( \text{LearnModel} \) infers one new local model using the newly executed test case \( t \) on the SUT. We use a simple linear parameter estimation method in line 6. The use of more powerful non-linear estimation methods here is an important topic of future research. Algorithm \( \text{UpdateModels} \) updates all the other existing local models using \( t \) and sorts the \( C \) best converged models.

In Algorithm 2 (line 6), model learning is performed using a linear parameter estimation method. The Mathematica kernel function \( \text{LinearSolve}(c, b) \) is used to find a vector of \((d + 1)^n\) coefficients satisfying the matrix equation \( c \cdot x = 1 \). We use Mathematica\textsuperscript{TM} version 7.0 running on a Linux platform.
Algorithm 1: Learning-BasedTesting

Input:
1. \( S(t: \text{TestCase}) \) - the SUT expressed as a function
2. \( \text{pre,post} \) - pre and postcondition for the SUT
3. \( d: \text{int} \) - the maximum approximation degree of \( \text{poly} \)
4. \( \text{max}: \text{int} \) - the maximum number of tests
5. \( C: \text{int} \) - the maximum number of model checks for each test

Output: \text{ErrorFound} or \text{TimeOut} if no error was found

// Initialisation phase
1. \( T, M \leftarrow \emptyset \) // Initialise set \( T \) of test cases and array \( M \) of models
2. \( n \leftarrow \text{input dimension of } S \)
3. \( \text{support}_\text{size} \leftarrow \text{pow}(d + 1,n) \)
4. \( T \leftarrow \text{set of } \text{support}_\text{size} \) randomly generated test cases
5. \text{foreach} \( t \) in \( T \) \text{do}
6. \( t.\text{output} \leftarrow S(t) \) // run \( S \) on \( t \)
7. \text{if} \( t \) violates \( \text{post} \) \text{then}
8. \text{return} \text{ErrorFound} + t.\text{toString()} \n
// Learning-based phase
9. \( \text{count} \leftarrow 0 \)
10. \text{while} \( \text{count} < \text{max} \) \text{do}
11. \( t \leftarrow \text{null} \)
12. \text{foreach} \( M[i] \) in \( M \) \text{do}
13. \( m \leftarrow \text{FindInstance}(m, \text{pre} \land \neg \text{post}) \)
14. \text{if} \( m.\text{converge} < \epsilon \) \text{then}
15. \( \text{Delete converged local model} \)
16. \( M.\text{delete}(m) \)
17. \text{if} \( t \) is NOT \text{null} \text{then}
18. \text{break} \n
19. \text{if} \( t \) is \text{null} \text{then}
20. \( t \leftarrow \text{FindInstance}(M[\text{random()}], \text{pre}) \)
21. \( t.\text{output} \leftarrow S(t) \) // run \( S \) on \( t \)
22. \text{if} \( t \) violates \( \text{post} \) \text{then}
23. \text{return} \text{ErrorFound} + t.\text{toString()} \n
24. \( M.\text{add}(\text{LearnModel}(T, t, \text{support}_\text{size})) \) // cf. Algorithm 2
25. \( T.\text{add}(t) \)
26. \( M \leftarrow \text{UpdateModels}(M, t, \text{support}_\text{size}, C) \) // cf. Algorithm 3
27. \( \text{count} \leftarrow \text{count} + 1 \)
28. \text{return} \text{TimeOut}
for the approximating \( d \) degree polynomial function in \( n \) variables. Note, if the call to \texttt{LinearSolve} fails, then temporarily we have no local model around \texttt{m.centrePoint}, but this can be corrected later by calls to Algorithm 3. Also note in Algorithm 2 that without any previous model history, we are not yet able to compute the convergence value of a newly constructed local model (as we do in Algorithm 3). Hence convergence is initialized to 0.0 (line 7). This forces the new local model to have the minimum possible convergence value, so it has the possibility to be satisfiability checked during the learning-based phase even though its convergence value is undefined.

In Algorithm 3, when updating a local model, a \textit{Monte-Carlo method} (lines 6, 7) is used to efficiently approximate the convergence value obtained using the integral norm \( L_1 \) on bounded polynomials. Each time local models are updated, the first \( C \) best converged local models are randomly swapped to the first \( C \) positions of the array \( M \), provided \( C \) is greater than or equal to the length of \( M \) (line 10, 11). This, together with line 14 in Algorithm 1, implements the prioritisation Step 3 of the abstract LBT algorithm of Section 2. In practice, the value of \( C \) is empirically determined. A higher value of \( C \) can probably take better advantage of model checking while it slows down the speed of learning-based testing if model checking is time consuming.

**Algorithm 2: LearnModel**

**Input:**
1. \( T \) - the set of all executed test cases
2. \( t \) : \texttt{TestCase} - a newly executed test case
3. \( \text{support\_size} \) : \texttt{int} - the number of test cases needed to build one local polynomial model

**Output:** a new local model with centre \( t \)

1. \( m \leftarrow \texttt{new\ LocalModel}() \)
2. \( m.centrePoint \leftarrow t \)
3. \( m.localPoints \leftarrow \text{from } T \text{ pick } \text{support\_size} \text{ test cases nearest to } t \)
4. \( m.radius \leftarrow \text{maximum Euclidean distance between } t \text{ and each data point of } m.localPoints \)
5. Form the matrix equation \( c \cdot x = b \) from \( m.localPoints \)
6. \( m.poly \leftarrow \texttt{LinearSolve}(c, b) \)
7. \( m.converg \leftarrow 0.0 \)
8. \textbf{return} \( m \)

## 4 Experimental Evaluation

### 4.1 Construction of SUTs, Specifications and Mutations

We wished to benchmark the performance of LBT against another iterative ATCG method for floating point computations. The simplest and most obvious
Algorithm 3: UpdateModels

Input:
1. \( M \) - array of all local models obtained so far
2. \( t \): TestCase - a newly executed test case
3. \( \text{support.size} \): int - cf. Algorithm 2
4. \( C \): int - cf. Algorithm 1

\[
\text{foreach LocalModel } m \text{ in } M \text{ do}
\]
\[
\text{if } t \text{ inside } m \text{ then}
\]
\[
T \leftarrow m.\text{localPoints.add}(t)
\]
\[
\hat{t} \leftarrow m.\text{centrePoint}
\]
\[
\hat{m} \leftarrow \text{LearnModel}(T, \hat{t}, \text{support.size})
\]
\[
\text{randomly pick } N \text{ test cases } t_1, t_2, \ldots, t_N
\]
\[
\hat{m}.\text{converg} \leftarrow \sum_{i=1}^{N} |m.\text{poly}(t_i) - \hat{m}.\text{poly}(t_i)|
\]
\[
m \leftarrow \hat{m}
\]
\[
\text{if } C < \text{length}(M) \text{ then}
\]
\[
\text{Partially sort } M \text{ to ensure } M[0 : C] \text{ are linearly ordered by convergence}
\]
\[
\text{Randomly permute } M[0 : C]
\]
\[
\text{else}
\]
\[
\text{Randomly permute } M
\]
\[
\text{return } M
\]

candidate for comparison was iterative random testing, which is fairly easy to implement when requirements specifications are simple. This comparison has the advantage that iterative random testing can be viewed as a Monte Carlo method to estimate the mean time to failure (MTF) of an SUT over an equiprobable distribution of input values. Our experiments confirmed that this estimated MTF value was inversely proportional to the size of injected mutation errors, as one would expect.

In order to obtain the largest possible data set of performance results, we used random generators to construct the numerical SUTs, their first-order logic specifications and their mutations. These also allowed us to perform experiments in a controlled way, in order to more accurately assess factors that influence the performance of our ATCG. The random numerical program generator (RPG) and specification generator (SG) were used to generate hundreds of SUTs with their specifications and introduce thousands of mutations. Iterative random testing was also repeated hundreds or thousands of times on each SUT, until the estimated MTF value appeared well converged.

Random Numerical Program Generation (RPG) To randomly generate a numerical program as an SUT, the RPG divides a global \( n \)-dimensional input space into subspaces at random. Within each of these randomly chosen subspaces, the RPG generates an \( n \)-dimensional polynomial surface of random shape (i.e. coefficients) and degree. We allowed for much higher degrees in such SUT models than in the learned models of Section 3.1 in order to ensure that the
learning problem would be non-trivial. When a random SUT was generated, we then mutated it randomly to inject faults by changing the shapes or positions of the polynomial surfaces in one or more subspaces. Figure 2 gives an example of a randomly generated SUT for one dimensional input and six randomly chosen subspaces. To mutate the SUT, we simply regenerate different curves over some of the same subspaces, and this is also shown in Figure 2. The ratio of the total size of the mutated subspaces to the total size of all subspaces represents the percentage error size. Controlling this percentage error size experimentally, was the key to understanding the relative performance of LBT and iterative random testing. For example in Figure 2, the percentage error size is 50%.

Random Specification Generation (SG) Since for evaluation purposes a large number of SUTs were automatically randomly generated, it was necessary to automatically generate their requirements specifications too. At the same time it was necessary to ensure that the requirement generated for each SUT was semantically correct in its unmutated state. It is well known (see [1]) that the logical complexity of requirements formulas has an effect on the efficiency of satisfiability checking, and hence on LBT as a whole. To explore the effects of this complexity, we studied the relative performance of LBT and IRT against two different logical types of requirements specifications.

Let $S$ be a randomly generated SUT with $k$ subspaces and 1-dimensional input $S = f_1(x), f_2(x), \ldots, f_k(x)$. Then for both types of requirements specifications, the same precondition was used. We call this an interval bound precondition on the input variable $x$ of the form

$$\text{pre}(S) \equiv c_1 \leq x \leq c_{k+1}.$$ 

Here the input interval $[c_1, c_{k+1}]$ for an SUT is divided into $k$ subintervals $[c_i, c_{i+1}]$ for $1 \leq i \leq k$ by the boundary values $c_1, c_2, \ldots, c_k$, and $f_i(x)$ describes the behaviour of $S$ over the $i$-th subinterval $[c_i, c_{i+1}]$.

On the other hand two different types of postconditions could be generated: we call these equational and inequational postconditions.
For the same SUT $S$, its \textit{equational postcondition} is a formula of the form:

$$\text{eq} \_ \text{post}(S) \equiv \bigwedge_{i=1,\ldots,k} (c_i \leq x < c_{i+1} \Rightarrow \|f_i(x) - m_i(x)\| < \epsilon)$$

where $m_i(x)$ describes the mutation of $f_i(x)$ (if any) over the $i$-th subinterval. Intuitively, $\text{eq} \_ \text{post}(S)$ asserts that the function of the mutated SUT is equal to the function of the original SUT $S$, to within an absolute tolerance $\epsilon$.

The \textit{inequational postcondition} for $S$ is a formula of the form:

$$\text{ineq} \_ \text{post}(S) \equiv \bigwedge_{i=1,\ldots,k} (\text{lower} \_ \text{bound} < f_i(x))$$

Intuitively, $\text{ineq} \_ \text{post}(S)$ asserts that all values of each function $f_i(x)$ lie above a constant lower bound. The value of $\text{lower} \_ \text{bound}$ is randomly chosen so that this postcondition is semantically correct for the unmutated (correct) program $S$.

These preconditions and two types of postcondition generalise in an obvious way to $n$-dimensional input for $n \geq 2$.

\subsection*{4.2 Results and Analysis}

Having set up random generators for numerical programs and their specifications, we proceeded to generate a large number of SUT/specification pairs, and mutate these SUTs to achieve different percentage error sizes within the range 10\% to 0.01\%. We then measured the minimum number of test cases using LBT and IRT needed to find the first true negative in each mutated SUT. This measurement was chosen since for IRT it provides an estimate of the mean time to failure (MTF) of the mutated SUT under an equiprobable input distribution. (We can view IRT as a Monte Carlo algorithm to estimate MTF.) To deal with the stochastic behavior of IRT, this value was averaged out over many runs until a well converged mean value had emerged. We then compared the ratio of these two measurements, and averaged out this figure over many different SUTs and many different mutations all of the same percentage error size. The results of our experiments are given in Figure 3 which illustrates the relative performance of LBT and IRT as the percentage error size is reduced. The $x$-axis expresses the percentage error size (c.f. Section 4.1) on a logarithmic scale. The $y$-axis gives the ratio IRT/LBT of the average number of IRT test cases divided by the average number of LBT test cases. Note that Figure 3 shows two curves, one for testing SUTs against equational specifications and one for testing SUTs against inequational specifications. Also note that above an error size of 10\%, both curves converge rapidly to $y = 1.0$, which explains our chosen range of error sizes.

The two distinct curves in Figure 3 clearly indicate that relative performance of LBT is influenced by the logical complexity of specifications, as we expected. However, the shapes of both curves are similar. Both curves show that as the
percentage error size decreases (or equivalently the MTF of the mutated SUT increases) the efficiency of LBT over IRT increases. Since the x-axis is logarithmic, this improvement in relative performance seems approximately exponential in the percentage size of errors.

4.3 A Concrete Case Study: Bubblesort

The statistical results of Section 4.2 may appear somewhat abstract, since the thousands of randomly generated case studies do not correspond to specific well known algorithms. Therefore, we complement this statistical analysis with a concrete case study.

```java
class BubbleSort {
    public void sort(double[] a) {
        for (int i = a.length; --i >= 0; ) {
            boolean flipped = false;
            for (int j = 0; j < i; j++) {
                // Mutated from "if (a[j] > a[j+1]) {
                if (a[j] > a[j+1]) {
                    double T = a[j];
                    a[j] = a[j+1];
                    a[j+1] = T;
                    flipped = true;
                    }
                if (!flipped) {
                    return;
                }
            }
        }
    }
}
```

Figure 4 presents the familiar Bubblesort algorithm for an array of floating point numbers. This algorithm represents a typical high dimensional problem,
since the input (and hence output) array size is usually rather large. In line 7 we introduce a mutation into the code via a parameter $N$. This particular mutation was mainly chosen to evaluate the quality of test cases, since it allows us to control the percentage error size of the mutation. Despite the high dimension of this SUT computation, pairwise LBT testing can find the mutation error fairly quickly. Figure 5 illustrates why this is so. Taking a 2-dimensional polynomial model on any output array value, Figure 5(a) shows that the SUT itself can be modeled quite easily. Furthermore Figure 5(b) shows that the mutated SUT can also be modeled with just a little more effort, since large regions of this model are again essentially simple. A suitable requirement specification for this code is just to assert that the output array is linearly ordered:

$$\{ \bigwedge_{i=0}^{a.length-1} MIN < a[i] < MAX \} \quad BubbleSort \quad \{ \bigwedge_{i=0}^{a.length-2} a[i] \leq a[i+1] \}$$

where $MIN$ and $MAX$ are simply the lower and upper bounds of input values.

As in Section 4.2, we can measure the minimum number of test cases required by LBT and IRT to find the first true negative in the mutated SUT, against the above requirement specification. Our results show that on average (since IRT has a stochastic performance) LBT is 10 times faster than IRT at uncovering the mutation error.

Fig. 5. Modeling the Bubblesort algorithm with and without mutation

5 Conclusion

We have presented a systematic and powerful extension of the learning-based testing (LBT) approach to iterative ATCG introduced in [10]. We have compared the performance of LBT against the results of iterative random testing over a large number of case studies. Our results clearly demonstrate that LBT,

---

2 Note that the grid structure in Figures 5(a) and 5(b) is an artifact of the Mathematica graphics package, the models themselves are non-gridded.
while never worse than iterative random testing, can be significantly faster at discovering errors. Future research will also consider non-linear models and learning algorithms for floating point data types. More generally, we also need to consider the problem of learned models, learning algorithms and satisfiability checkers for other data types besides floating point, in order to increase the range of applicability of our testing methods.

Acknowledgement

We gratefully acknowledge financial support for this research from the Chinese Scholarship Council (CSC), the Swedish Research Council (VR) and the European Union under project HATS FP7-231620.

References

Learning-Based Testing for Reactive Systems using Term Rewriting Technology

K. Meinke, F. Niu
School of Computer Science and Communication, Royal Institute of Technology, 100-44 Stockholm, Sweden, karlm@nada.kth.se, niu@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 using term rewriting technology. A general model for a reactive system can be given by an extended Mealy automata (EMA) over an abstract data type (ADT). A finite state EMA over an ADT can be efficiently learned in polynomial time using the CGE regular inference algorithm, which builds a compact representation as a complete term rewriting system. We show how this rewriting system can be used to model check the learned automaton against a temporal logic specification by means of narrowing. Combining CGE learning with a narrowing model checker we obtain a new and general architecture for learning-based testing of reactive systems. We compare the performance of this LBT architecture against random testing using a case study.

1 Introduction

Learning-based testing (LBT) is an emerging technology for specification-based black-box testing that encompasses the three essential steps of: (1) test case generation (TCG), (2) test execution, and (3) test verdict (the oracle step). It has been successfully applied to testing procedural systems in [13] and [15], and reactive systems in [16]. 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 algorithm, and integrating these two with the system under test (SUT) in an iterative loop. The use of incremental learning is critical in making this technology both fast and scalable to large systems under test (SUTs). Our previous research ([13] and [16]) has repeatedly shown that LBT has the capability to significantly outperform random testing in the speed with which it finds errors in an SUT.

For testing complex embedded software systems, there is a significant need to generate test cases over infinite data types such as integer and floating point types, and abstract data types (ADTs) such as strings, arrays, lists and various symbolic data types. Specification-based TCG is essentially a constraint solving problem. So this generalisation from finite to infinite and symbolic data types is highly non-trivial since the satisfiability problem for many logics over abstract
and infinite data types is undecidable. Thus a search for test cases is not guaranteed to terminate.

Model checking over abstract and infinite data types is therefore a state of the art problem. Recently some success has been achieved with the use of satisfiability modulo theories (SMT) solvers such as Z3 [5], which are based on heuristic techniques. However, an alternative approach is to use constraint solving based on a narrowing algorithm. Narrowing is a flexible technology, based on term rewriting, which is applicable to any data type for which we can find a complete (confluent and terminating) term rewriting system (see e.g. [1]). It has well understood theoretical properties such as completeness of solutions and conditions for termination. Narrowing has been successfully applied to model checking of infinite state systems in [6]. However, the use of narrowing for test case generation has not yet been considered. In fact, our aim in this paper is much wider. We will show that narrowing combines easily with symbolic learning algorithms for automata such as CGE [14] to yield a new LBT architecture for specification-based testing of reactive systems computing over abstract data types.

Initial case studies suggest that despite the significant increase in the problem complexity, this new LBT architecture is also competitive with random testing.

The structure of this paper is as follows. In the remainder of Section 1 we review related work. In Section 2, we recall some essential mathematical preliminaries needed to discuss narrowing. In Section 3, we formalise a general model of a reactive system as an extended Mealy automaton (EMA) over an abstract data type (ADT). We introduce a linear time temporal logic (LTL) for such EMA, and we show how an LTL formula can be translated into constraint sets consisting of equations and negated equations. In Section 4, we present a model checking algorithm based on narrowing applied to a constraint set. In Section 5, we combine this model checking method with a symbolic automata learning algorithm (the CGE learning algorithm of [14]) to define a new LBT architecture for specification-based testing of reactive systems. In Section 6, we present a case study of this LBT architecture applied to testing the TCP protocol. Finally, in Section 7 we draw some conclusions and discuss open questions to be addressed by future work.

1.1 Related Work

In [16], LBT was applied to testing reactive systems modeled as Boolean Kripke structures. Our work here extends this previous work to allow symbolic and infinite data types. Even for finite data types, this approach simplifies the expression of control and data properties of an SUT. For this extension we use a more powerful symbolic learning algorithm, new model checking technology based on term rewriting theory, and a more powerful oracle construction for test verdicts.

Several previous studies, (for example [19], [9] and [20]) 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 (CE-
GAR) also combines learning and model checking, (see e.g. [3] and [2]). The LBT approach described here can be distinguished from these other approaches by: (i) an emphasis on testing rather than verification, (ii) the focus on incremental learning for efficient scalable testing, (iii) the use of narrowing as a model checking technique, and (iv) the introduction of abstract data types.

There is of course an extensive literature on the use of model checkers (without learning) to generate test cases for reactive systems. A recent survey is [8]. Generally this work emphasizes glass-box testing (so no learning is necessary), and the use of structural coverage measures to constrain the search space for test cases. Furthermore, behavioral requirements may or may not be present. By contrast, the LBT approach concerns black-box testing. Furthermore, in LBT behavioral requirements are always present, both to solve the oracle problem and to constrain the search space and guide the search for effective test cases.

In [21], black-box reactive system testing using learning but without model checking is considered. This is also shown to be more effective than random testing. Thus we can conclude that learning and model checking are two mutually independent techniques that can be applied to systems testing separately or together. In the long term we hope to show that the combination of both techniques is ultimately more powerful than using either one alone.

## 2 Mathematical Preliminaries and Notation

It is helpful to have some familiarity with the theories of abstract data types and term rewriting. Both use the notation and terminology of many-sorted algebra (see e.g. [17]). Let $S$ be a finite set of sorts or types. An $S$-sorted signature $\Sigma$ consists of an $S^* \times S$-indexed family of sets $\Sigma = (\Sigma_{w,s} | w \in S^*, s \in S)$. For the empty string $\epsilon \in S^*$, $c \in \Sigma_{\epsilon,s}$ is a constant symbol of sort $s$. Let $w = s_1, \ldots, s_n \in S^+$, $f \in \Sigma_{w,s}$ is a function symbol of arity $n$, domain type $w$ and codomain type $s$. An $S$-sorted $\Sigma$-algebra $A$ consists of sorts, constants and functions that interpret $\Sigma$ by a particular semantics. Thus $A$ has an $S$-indexed family of sets $A = \{A_s | s \in S\}$, where $A_s$ is termed the carrier set of sort $s$. For each $s \in S$ and constant symbol $c \in \Sigma_{\epsilon,s}$, $c_A \in A_s$ is a constant, and for each $w = s_1, \ldots, s_n \in S^+$ and each $f \in \Sigma_{w,s}$, $f_A : A_{s_1} \times \cdots \times A_{s_n} \rightarrow A_s$ is a function. Let $X = (X_s | s \in S)$ be an $S$-indexed family of disjoint sets $X_s$ of variables of sort $s$. We assume $X_s \cap \Sigma_{\epsilon,s} = \emptyset$. The set $T(\Sigma, X)_s$ of all terms of sort $s \in S$ is defined inductively by: (i) $c \in T(\Sigma, X)_s$ for $c \in \Sigma_{\epsilon,s}$, (ii) $x \in T(\Sigma, X)_s$ for $x \in X_s$, and (iii) $f(t_1, \ldots, t_n) \in T(\Sigma, X)_s$ for $f \in \Sigma_{w,s}$ with $w = s_1, \ldots, s_n$ and $t_i \in T(\Sigma, X)_{s_i}$ for $1 \leq i \leq n$. We use $\equiv$ to denote syntactic equality between terms. An equation $e$ (respectively negated equation) over $\Sigma$ and $X$ is a formula of the form $(t = t')$ (respectively $\neg(t = t')$) for $t, t' \in T(\Sigma, X)_s$. We let $\text{Vars}(t)$ (respectively $\text{Vars}(e)$, $\text{Vars}(\neg(e))$) denote the set of all variables from $X$ occurring in $t$ (respectively $e$, $\neg e$).

A variable assignment $\alpha : X \rightarrow A$ is an $S$-indexed family of mappings $\alpha_s : X_s \rightarrow A_s$. A substitution $\sigma$ is a variable assignment $\sigma : X \rightarrow T(\Sigma, X)$ such that $\sigma_s(x) \neq x$ for just finitely many $s \in S$ and variables $x \in X_s$, and this
set of variables is the domain of $\sigma_s$. The result of applying a substitution $\sigma$ to a term $t \in T(\Sigma, X)_s$ is defined inductively in the usual way and denoted by $\sigma(t)$. If $\sigma$ and $\sigma'$ are substitutions then their composition $\sigma \circ \sigma'$ is defined by $\sigma \circ \sigma'(x) = \sigma(\sigma'(x))$. A variable renaming is a family of bijective substitutions $\sigma_s : X_s \rightarrow X_s$. A substitution $\sigma$ is more general than a substitution $\tau$, denoted $\sigma \leq \tau$ if there exists a substitution $\delta$ such that $\delta \circ \sigma = \tau$.

A disunification problem $S = \{ (t_i, Q_i, t'_i) \mid i = 1, \ldots, n, n \geq 0, Q_i \in \{ \neq, \neq \} \}$ is a finite (possibly empty) set of equations and negated equations over $\Sigma$ and $X$. A substitution $\sigma : X \rightarrow T(\Sigma, X)$ is a syntactic unifier of a set $S$ if for all $1 \leq i \leq n$, if $Q_i$ is $\neq$ then $\sigma(t_i) \equiv \sigma(t'_i)$, and if $Q_i$ is $\neq$ then $\sigma(t_i) \neq \sigma(t'_i)$. We let $U(S)$ denote the set of all syntactic unifiers of $S$. A unifier $\sigma \in U(S)$ is a most general unifier (mgu) if $\sigma \leq \tau$ for all $\tau \in U(S)$.

If $t \in T(\Sigma, X)_s$ is a term then $O(t)$ denotes the set of all positions in $t$, i.e. all nodes in the parse tree of $t$ and is inductively defined by $O(\epsilon) = O(x) = \{ \epsilon \}$ and $O(f(t_1, \ldots, t_n)) = \{ \epsilon, k.i \mid 1 \leq k \leq n, i \in O(t_k) \}$. We write $t|_p$ for the subterm of $t$ found at position $p \in O(t)$, and if $t|_p$, $u \in T(\Sigma, X)_s$ then $t[u]_p$ denotes the term obtained by replacing the subterm found at $p$ in $t$ by $u$. We say that $p \in O(t)$ is a non-variable position if $t|_p$ is not a variable, and let $\overline{O}(t)$ denote the set of all such non-variable positions.

A term rewriting rule is an expression of the form $l \rightarrow r$ for $l, r \in T(\Sigma, X)_s$ and $s \in S$ such that $\text{Vars}(r) \subseteq \text{Vars}(l)$ and a term rewriting system (TRS) $R$ is a set of rewriting rules. If $\sigma_s : X_s \rightarrow X_s$ is a family of variable renamings then $\sigma(l) \rightarrow \sigma(r)$ is a variant of $l \rightarrow r$. The rewrite relation $\xrightarrow{R_s} \sigma(l)$ associated with a TRS $R$ is a binary relation on terms defined by $t \xrightarrow{R_s} t'$ if there exists a rule $l \rightarrow r \in R$, a position $p \in O(t)$ and a substitution $\sigma$ such that $t|_p \equiv \sigma(l)$ and $t' \equiv t(\sigma(r))_p$. We call $t \xrightarrow{R_s} t'$ a rewrite step. We let $\xrightarrow{R_s}^*$ denote the reflexive transitive closure of $\xrightarrow{R_s}$. A TRS $R$ is strongly transitive if there is no infinite sequence of rewrite steps $t_0 \xrightarrow{R_s} t_1 \xrightarrow{R_s} t_2 \xrightarrow{R_s} \ldots$ and $R$ is confluent (or Church-Rosser) if for any terms $t, t_1, t_2 \in T(\Sigma, X)_s$, if $t \xrightarrow{R_s} t_1$ and $t \xrightarrow{R_s} t_2$ then there exists $t' \in T(\Sigma, X)_s$ such that $t_1 \xrightarrow{R_s} t'$ and $t_2 \xrightarrow{R_s} t'$. A complete TRS is confluent and strongly transitive.

3 Mealy Automata over Abstract Data Types

In this section we formalise a general model of a reactive system as an extended Mealy automaton (EMA) over an abstract data type. We then introduce the syntax and semantics of a linear time temporal logic (LTL) as a language for expressing user requirements on EMA. Finally, we define a syntactic translation of LTL into equations and negated equations, and establish the soundness and completeness of this translation with respect to satisfiability.

We can model a Mealy automaton over an abstract data type as a many-sorted algebraic structure by considering inputs, states and outputs as distinguished data sorts (or types). The input and output types will be typically chosen
from some well-known data types such as \textit{int}, \textit{string}, \textit{array}, \textit{list} etc. that provide a high level of data abstraction.

\textbf{Definition 1.} A signature \((S, \Sigma, \text{input}, \text{output})\) for an extended Mealy automaton is a four-tuple, where \(S = \{\text{state}, s_1, \ldots, s_n\}\) is a sort set, \(\Sigma\) is an \(S\)-sorted signature with distinguished constant and function symbols

\[ q^0 \in S_{\text{state}}, \delta \in \Sigma_{\text{state input state}}, \lambda \in \Sigma_{\text{state input output}}, \]

and \(\text{input}, \text{output} \in \{s_1, \ldots, s_n\}\) are distinguished input and output types.

\textbf{Definition 2.} Let \((S, \Sigma, \text{input}, \text{output})\) be a signature for an EMA. An extended Mealy automaton \(A\) (of signature \(\Sigma\)) is an \(S\)-sorted \(\Sigma\)-algebra \(A\).

As usual \(q^0_A\) is the initial state, \(\delta_A : A_{\text{state}} \times A_{\text{input}} \rightarrow A_{\text{state}}\) is the state transition function, and \(\lambda_A : A_{\text{state}} \times A_{\text{input}} \rightarrow A_{\text{output}}\) is the output function.

We define the extended state transition and output functions

\[ \delta^*_A : A_{\text{state}} \times A^*_{\text{input}} \rightarrow A_{\text{state}}, \quad \lambda^*_A : A_{\text{state}} \times A^+_{\text{input}} \rightarrow A_{\text{output}}\]

in the usual way for any \(q \in A_{\text{state}}, i \in A_{\text{input}}\) and \(j \in A_{\text{input}}\) by \(\delta^*_A(q, i) = q\) and \(\lambda^*_A(q, i) = \lambda_A(\delta_A(q, i), j)\).

If \(A_{\text{state}}\) is finite then \(A\) is termed a finite state EMA, otherwise \(A\) is termed an infinite state EMA.

Next we introduce a linear-time temporal logic (LTL) that can be used to express user requirements on EMA. For this it is necessary to integrate the underlying data type signature \(\Sigma\) in an appropriate way. In the sequel we assume that \((S, \Sigma, \text{input}, \text{output})\) is a given EMA signature. Let \(X = \{X_s \mid s \in S - \{\text{state}\}\}\) be any indexed family of sets \(X_s\) of variable symbols of sort \(s\). We assume that \(\text{input} \in X_{\text{input}}\) and \(\text{output} \in X_{\text{output}}\) are two distinguished variable symbols.

\textbf{Definition 3.} The set \(\text{LTL}(\Sigma, X)\) of all linear temporal logic formulas over \(\Sigma\) and \(X\) is defined to be the smallest set of formulas containing the atomic proposition \textit{true} and all equations \((t = t')\) for each sort \(s \in S - \{\text{state}\}\) and all terms \(t, t' \in T(\Sigma, X)\), which is closed under negation \(\neg\), conjunction \(\land\), disjunction \(\lor\), and the next \(X\), always future \(G\), sometime future \(F\), always past \(G^{-1}\), and sometime past \(F^{-1}\) temporal operators

As usual, \(X(\phi)\) denotes that \(\phi\) is true in the next time instant, while \(G(\phi)\) (respectively \(F(\phi)\)) denotes that \(\phi\) is always (respectively at some time) true in the future of a run. On the other hand \(G^{-1}(\phi)\) (respectively \(F^{-1}(\phi)\)) denotes that \(\phi\) was always (respectively at some time) true in the past of a run. While not strictly necessary, including these past operators makes this LTL exponentially more succinct, as shown in [12]. This increases the efficiency of our narrowing model checker. We let \((\phi \implies \psi)\) denote the formula \((\neg \phi \lor \psi)\), and \(t \neq t'\) denotes \(\neg(t = t')\). Then for example, the formula

\[ G(\text{in} = x) \land X(\text{in} = y) \implies X(\text{out} = x + y) \]
is an LTL formula that expresses that at all times, if the current input is \( x \) and
next input is \( y \) then in two time steps from now the output will be the sum
\( x + y \). So in this LTL we can express both control and data properties of reactive
systems.

**Definition 4.** Let \( A \) be an EMA, let \( n \in \mathbb{N} \), let \( \bar{i} = i_0, i_1, \ldots \in A_{\text{input}}^\omega \) be
an infinite sequence of inputs for \( A \), and let \( \text{Val}_{A,\alpha} : T(\Sigma, X)_n \to A \) be the
valuation mapping on terms given a variable assignment \( \alpha : X \to A \). We define the
saturation relation \( \models \) as follows:

(i) \( A, n, \bar{i}, \alpha \models \top \) if and only if, \( \text{Val}_{A,\alpha}(t) = \text{Val}_{A,\alpha}(t') \), where
\[
\beta = \alpha[ \text{in} \mapsto i_n, \text{out} \mapsto \lambda(\delta^\ast(q_0, i_0, \ldots, i_{n-1}), i_n) ].
\]

(ii) \( A, n, \bar{i}, \alpha \models t = t' \) if and only if, \( \text{Val}_{A,\beta}(t) = \text{Val}_{A,\beta}(t') \), where
\[
\beta = \alpha[ \text{in} \mapsto i_n, \text{out} \mapsto \lambda(\delta^\ast(q_1, i_0, \ldots, i_{n-1}), i_n) ].
\]

As is well known, for every formula \( \phi \in \text{LTL}(\Sigma, X) \) there exists a logically equivalent
formula \( \phi' \in \text{LTL}(\Sigma, X) \) in negation normal form (NNF) where negations
only occur in front of atomic subformulas. To solve LTL formulas by narrowing
we translate an NNF formula \( \phi \) into a finite set \( S = \{ S_1, \ldots, S_n \} \) of constraint
sets, where a constraint set \( S_i \) consists of equations and negated equations. This
translation requires an additional set \( X = \{ x_i | x_i \in X_{\text{input}} \} \) of fresh variable
symbols ranging over input sequence elements.

**Definition 5.** Let \( A \) be an EMA, and let loopbound be the length of the longest
loop-free path in \( A \). For each NNF formula \( \phi \in \text{LTL}(\Sigma, X) \) we define the
satisfaction set \( \text{SatSet}_n(\phi) \) as a finite collection of constraint sets by structural
induction on \( \phi \).

\[
\text{SatSet}_n(\top) = \{ \} \quad \text{SatSet}_n(t \top) = \{ \{ \theta_n(t) Q \theta_n(t') \} \}
\]

where \( Q \in \{ =, \neq \} \) and \( \theta_n \) is the substitution defined by
\[
\theta_n = \{ \text{in} \mapsto x_n, \text{out} \mapsto \lambda(\delta^\ast(q_0, x_0, \ldots, x_{n-1}), x_n) \}
\]
The translation \( \text{SatSet}_n(\phi) \) preserves solutions of \( \phi \) as follows.

**Theorem 1.** Let \( A \) be an EMA, and \( \text{loopbound} \) be the length of the longest loop-free path in \( A \). Let \( \phi \in \text{LTL}(\Sigma, X) \) be in NNF, and let \( n \in \mathbb{N} \).

(i) (Soundness of Translation) For any assignment \( \alpha : X \to A \) and input sequence \( \overline{i} = i_0, i_1, \ldots \in A^\omega_{\text{input}} \) there exists \( S \in \text{SatSet}_n(\phi) \) such that

\[
A, n, \overline{i}, \alpha \models \phi \implies A, \beta(\overline{i}), \alpha \models S,
\]

where the assignment \( \beta(\overline{i}) : X \to A_{\text{input}} \) is given by \( \beta(\overline{i})(\overline{x}) = i_n \).

(ii) (Completeness of Translation) For any assignments \( \alpha : X \to A \) and \( \beta : X \to A_{\text{input}} \) if there exists \( S \in \text{SatSet}_n(\phi) \) such that \( A, \beta, \alpha \models S \) then there exists an input sequence \( \overline{\beta} \in A^\omega_{\text{input}} \) such that

\[
A, n, \overline{\beta}, \alpha \models \phi.
\]

Thus by Theorem 1, to solve an NNF formula \( \phi \) it is necessary and sufficient to solve one of the constraint sets \( S_1, \ldots, S_n \in \text{SatSet}_0(\phi) \). We will consider a method to solve constraint sets by narrowing in the next section.
4 Model Checking by Narrowing

The problem of finding solutions to a set \{ t_1 = t'_1, \ldots, t_n = t'_n \} of equations is the well known unification problem about which much has been written (see e.g. [1]). More generally, in the case that a set \{ t_1 = t'_1, \ldots, t_n = t'_n, u_1 \neq u'_1, \ldots, u_n \neq u'_n \} of equations and negated equations must be solved, this problem is known as the disunification problem (see e.g. [4]).

Let \( \Sigma \) be a many-sorted data type signature and \( E \) be an equational data type specification having a complete rewrite system \( R \). Then the disunification problem is complicated by the fact that we seek solutions modulo \( R \) (and hence \( E \)) in the following sense.

**Definition 6.** Let \( R \) be a term rewriting system. The relation of \( R \)-conversion denote by \( =_R \) is the reflexive symmetric and transitive closure of \( \xrightarrow{R} \). Let \( S = \{ (t_i, Q_i, t'_i) \mid i = 1, \ldots, n, n \geq 0, Q_i \in \{ =, \neq \} \} \) be a disunification problem. A substitution \( \sigma : X \rightarrow T(\Sigma, X) \) is an \( R \)-unifier of \( S \) if for all \( 1 \leq i \leq n \), if \( Q_i = \) then \( \sigma(t_i) =_R \sigma(t'_i) \), and if \( Q_i = \neq \) then \( \sigma(t_i) \neq_R \sigma(t'_i) \). We let \( \mathcal{U}_R(S) \) denote the set of all \( R \)-unifiers of \( S \).

In the special case where \( E = R = \emptyset \), these problems are known as syntactic unification and syntactic disunification, and both problems are decidable. However in many important cases, both the unification and disunification problems are undecidable. Nevertheless, these problems are semidecidable and one can consider complete search algorithms which always terminate when a solution is to be found. The method of narrowing gives such a complete search algorithm, and can be used whenever the data type specification \( E \) can be represented by a complete term rewriting system \( R \).

The basic idea of narrowing is a systematic search of the space of possible solutions using the rules of \( R \). If some equation \( t_i = t'_i \) cannot be syntactically unified then we can apply a substitution \( \sigma : X \rightarrow T(\Sigma, X) \) to \( t_i \) (or \( t'_i \)) such that the resulting term \( \sigma(t_i) \) is not in \( R \) normal form and then reduce this in one step. This requires unifying \( t_i \) (or \( t'_i \)) with the left hand side \( l \) of a rule \( l \rightarrow r \) in \( R \), and replacing with a suitable instance of \( r \) so that a new equation is obtained. A similar process can be applied to negated equations, and can be iterated for all formulas until syntactic unification of the entire set becomes possible, though the narrowing process may not terminate. If it terminates, the resulting sequence of substitutions \( \sigma_k : X \rightarrow T(\Sigma, X) \) can be composed together with the final syntactic unifier \( \theta \) to yield an \( R \)-unifier.

**Definition 7.** We say that a term \( t \) is \( R \)-narrowable into a term \( t' \) if there exists a non-variable position \( p \in \overline{O}(t) \), a variant \( l \rightarrow r \) of a rewrite rule in \( R \) and a substitution \( \sigma \) such that:

(i) \( \sigma \) is a most general syntactic unifier of \( t|_p \) and \( l \), and
(ii) \( t' \equiv \sigma(t|_p) \).

We write \( t \xrightarrow{\sigma|_p, l \rightarrow r, \sigma} t' \) or simply \( t \xrightarrow{\sigma} t' \). The relation \( \xrightarrow{\sigma} \) is called \( R \)-narrowing.
The $R$-narrowing relation on terms can be extended to equations and negated equations in an obvious way. A formula $(t \, Q \, t')$ (where $Q$ is = or $\neq$) is $R$-narrowable into a formula $(u \, Q \, u')$ if there exists a variant $l \to r$ of a rewrite rule in $R$ and a substitution $\sigma$ such that either $t \sim_{[p, l \to r, \sigma]} u$ for some non-variable occurrence $p \in \mathcal{T}(t)$ or $t' \sim_{[p, l \to r, \sigma]} u'$ for some non-variable occurrence $q \in \mathcal{T}(t')$. We write $(t \, Q \, t') \sim_{[p, l \to r, \sigma]} (u \, Q \, u')$ or simply $(t \, Q \, t') \sim_{\sigma} (u \, Q \, u')$. Generalising the $R$-narrowing relation still further to sets of equations and negated equations we will write $S \sim_{[p, l \to r, \sigma]} S'$ or $S \sim_{\sigma} S'$.

We can relativise the concept of a substitution $\sigma$ being more general than a substitution $\tau$ (c.f. Section 2) to $R$ as follows. Let $V$ be any $S$-indexed family of sets $V_s$ of variables. We define $\sigma \leq_R \tau[V]$ if for some substitution $\delta$, $\delta \circ \sigma(x) = \tau(x)$ for all $s \in S$ and $x \in V_s$. Now we can discuss the soundness and completeness of narrowing.

**Theorem 2.** Let $S = \{ (t_i, Q_i, t'_i) \mid i = 1, \ldots, n, n \geq 0, Q_i \in \{ =, \neq \} \}$ be a disunification problem.

(i) (Soundness of Narrowing) Let $S \sim_{\sigma_1} S_1 \sim_{\sigma_2} \ldots \sim_{\sigma_n} S_n$ be a terminated $R$-narrowing derivation such that $S_n$ is syntactically unifiable by a substitution $\theta$. Then $\theta \circ \sigma_n \circ \ldots \circ \sigma_1$ is an $R$-unifier of $S$.

(ii) (Completeness of Narrowing) If $S$ is $R$-unifiable then let $\rho$ be any $R$-unifier and $V$ be a finite set of variables containing $\text{Vars}(S)$. There exists a terminated $R$-narrowing derivation $S \sim_{\sigma_1} S_1 \sim_{\sigma_2} \ldots \sim_{\sigma_n} S_n$ such that $S_n$ is syntactically unifiable. Let $\mu$ be a most general syntactic unifier of $S_n$ then $\mu \circ \sigma_n \circ \ldots \circ \sigma_1 \leq_R \rho[V]$.

The search space of narrowing is large and narrowing procedures frequently fail to terminate. Many proposals have been made to increase the efficiency of narrowing. One important restriction on the set of occurrences available for narrowing, termed basic narrowing, was introduced in [11], and has since been widely studied, e.g. [18].

A basic narrowing derivation is very similar to a narrowing derivation as given in Definition 7 above. However, in a basic narrowing derivation, narrowing is never applied to a subterm introduced by a previous narrowing substitution. This condition is quite complex to define precisely, and the reader is referred to [11].

Theorem 4 of [11] can be used to show that basic narrowing for equations and negated equations is also sound and complete in the sense of Theorem 2. However, for basic narrowing [11] also establishes sufficient conditions to guarantee termination. This property is important in test case generation, where we need to know if a test case exists at all.
Theorem 3. ([11]) Let \( R = \{ l_i \rightarrow r_i \mid i = 1, \ldots, n \} \) be a complete rewrite system such that any basic \( R \)-narrowing derivation from any of the \( r_i \)'s terminates. Then every \( R \)-narrowing derivation terminates.

Many examples of TRS satisfying Theorem 3 are known, including TRS for all finite ADTs. This general termination result can be applied to establish that basic \( R \)-narrowing yields a decision procedure for LTL model checking (i.e. basic \( R \)-narrowing is sound, complete and terminating) because of the following new result about the CGE symbolic learning algorithm.

Theorem 4. Let \((R_n^{\text{state}}, R_n^{\text{output}})\) be the output of the CGE learning algorithm after a sequence of \( n \) observations of the I/O behavior of an EMA \( A \). Then \( R_n = R_n^{\text{state}} \cup R_n^{\text{output}} \) is a complete rewrite system and every \( R_n \)-narrowing derivation terminates.

Proof. Proposition 4.5 of [14] establishes that \( R_n \) is complete. To establish termination, consider that every rule \( l \rightarrow r \in R_n \) is ground by Definitions 3.12 and 4.4 of [14]. Hence the result is a special instance of Theorem 3 above and Example 3 in [11].

We have constructed an implementation of model checking by basic narrowing. We explain how this is integrated into learning-based testing in Section 5.

5 An LBT Architecture for Testing Reactive Systems

Learning-based testing (LBT) is a general paradigm for black-box specification-based testing that requires three basic 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 \( S \).

Given such components, the paradigm provides a heuristic iterative method to search for and automatically generate a sequence of test cases. The basic idea is to incrementally learn an approximating sequence of models \( M_i \) for \( i = 1, 2, \ldots \) of the unknown SUT \( S \) by using test cases as queries. During this learning process, we model check each approximation \( M_i \) on-the-fly searching for counterexamples to the validity of \( \text{Req} \). Any such counterexample can be confirmed as a true negative by taking it as the next test case. At step \( i \), if model checking does not produce any counterexamples then to proceed with the iteration, the next test case is constructed by another method, e.g. randomly.

In [16], LBT was applied to testing reactive systems modeled as Boolean Kripke structures. In this paper we consider the case where the SUT \( S \) is a reactive system that can be modeled by an EMA over the appropriate abstract data types, and \( \text{Req} \) is an LTL formula over the same data types. Thus we extend the scope of our previous work to deal with both control and data by applying new learning algorithms and model checking technology.
For LBT to be effective at finding errors quickly, it is important to use an incremental learning algorithm. In [16] this was empirically demonstrated by using the IKL incremental learning algorithm for Boolean Kripke structures. However, learning algorithms for finite data types such as IKL do not extend to infinite data types. The CGE learning algorithm of [14] was designed to implement learning EMA over abstract data types. Furthermore, this algorithm is incremental since its output is a sequence of representations $R_1, R_2, \ldots$ of the hypothesis EMA $M_1, M_2, \ldots$ which are the approximations to $S$. Each representation $R_i$ is a complete TRS that encodes $M_i$ as the corresponding quotient of the prefix tree automaton. Details of this representation can be found in [14]. Furthermore, CGE has many technical advantages over IKL. For example, the number of queries (test cases) between construction of successive approximations $R_k$ and $R_{k+1}$ can be arbitrarily small and even just one query. By contrast, IKL and other table based learning algorithms usually have intervals of tens or hundreds of thousands of queries between successive approximations of large SUTs.

As a consequence, model checking can only be infrequently applied.

The input to CGE is a series of pairs $(i_1, o_1), (i_2, o_2), \ldots$ consisting of a query string $i_k$ for $S$ and the corresponding output string $o_k$ from $S$. In an LBT setting, the query strings $i_k$ come from model checker counterexamples and random queries. Finite convergence of the sequence $R_1, R_2, \ldots$ to some TRS $R_n$ can be guaranteed if $S$ is a finite state EMA (see [14]) and the final hypothesis automaton $M_n$ is behaviorally equivalent with $S$. So with an increasing number of queries, it becomes more likely that model checking will produce a true negative if one exists, as the unknown part of $S$ decreases to nothing. By combining CGE with the narrowing model checker of Section 4, we arrive at a new LBT architecture for reactive systems shown in Figure 1.

Figure 1 illustrates the basic iterative loop of the LBT architecture between: (i) learning, (ii) model checking, (iii) test execution, and (iv) test verdict by an oracle. This iterative loop is terminated by an equivalence checker. This component can be used to detect that testing is complete when the SUT is sufficiently small to be completely learned. Obviously testing must be complete by the time we have learned the entire SUT, since model checking by narrowing is solution complete. The equivalence checker compares the current model representation $R_k$ with $S$ for behavioural (rather than structural) equivalence. A positive result from this equivalence test stops all further learning, after one final model check of $R_k$ searches for any residual errors. In practical applications of LBT technology, real world SUTs are usually too large to be completely learned. It is this pragmatic constraint that makes incremental learning algorithms necessary for scalable LBT. In such cases the iterative loop must ultimately be terminated by some other method such as a time constraint or a coverage measure.

Figure 1 shows that the current model $R_k$ is also passed from the CGE algorithm to the basic narrowing model checker, together with a user requirement represented as an LTL formula $\phi$. This formula is fixed for a particular testing session. The model checker uses $R_k$ to identify at least one counterexample to $\phi$ as an input sequence $i_{k+1}$ over the underlying input data type. If $\phi$ is a safety
formula then this input sequence will usually be finite
\[ \overline{i_{k+1}} = (i_1, \ldots, i_j) \in \mathcal{T}(\Sigma)_{\text{input}}^*. \]

If \( \phi \) is a liveness formula then the input sequence \( \overline{i_{k+1}} \) may be finite or infinite. Since infinite counterexamples to liveness formulas can be represented as infinite strings of the form \( \pi \overline{y}^\omega \), in this case \( \overline{i_{k+1}} \) is truncated to a finite initial segment that would normally include at least one execution of the infinite loop \( \overline{y}^\omega \), such as \( \overline{i_{k+1}} = \pi \overline{y} \). Observing the failure of infinite test cases is of course impossible, and the LBT architecture implements a compromise solution that executes the truncated input sequence only, and issues a warning rather than a definite test failure.

![Diagram of the LBT architecture](image)

**Fig. 1.** A Learning-based Testing Architecture for Reactive Systems

If the next test case \( \overline{i_{k+1}} \) cannot be constructed by model checking then in order to proceed with iterative testing a random input string generator (see Figure 1) is used to generate \( \overline{i_{k+1}} \). During this random generation process, any random string that has been used as a previous test case is discarded to avoid redundant replicate tests.

Thus from one of two possible sources (model checking or random generation) a new test case \( \overline{i_{k+1}} \) is constructed. Figure 1 shows that this new test case \( \overline{i_{k+1}} \) is then executed on the SUT \( S \) to yield an actual output sequence \( \overline{o_{k+1}} = o_1, \ldots, o_j \). The pair \( (\overline{i_{k+1}}, \overline{o_{k+1}}) \) is then passed to an oracle to compute the \( k+1 \)-th test verdict.

The oracle we have developed for this LBT architecture is more powerful than the one described in [16], and is based on the following two step process.
Step 1. A test verdict can often be derived quickly and simply by computing a predicted output \( p_{k+1} = p_1, \ldots, p_j \) obtained by simulating the behavior of \( M_k \) on \( i_{k+1} \). This is easily derived by applying the TRS \( R_k \) to rewrite the input string \( i_{k+1} \) into its normal form, i.e. \( i_{k+1} \xrightarrow{R_k} p_{k+1} \). Recall that \( R_k \) is a complete TRS, so this normal form is always well defined. We then implement a simple Boolean test \( o_{k+1} = p_{k+1} \). If this equality test returns true and the test case \( i_{k+1} \) was originally a finite test case then we can conclude that the test case \( i_{k+1} \) is definitely failed, since the behavior \( p_{k+1} \) is by construction a counterexample to the correctness of \( \phi \). In this case we can decide to stop testing. If the equality test returns true and the test case \( i_{k+1} \) was finitely truncated from an infinite test case (a counterexample to a liveness requirement) then the verdict is weakened to a warning (but testing is not stopped). This is because the most we can conclude is that we have not yet seen any difference between the observed behavior \( o_{k+1} \) and the incorrect behavior \( p_{k+1} \).

Step 2. If the Boolean test \( o_{k+1} = p_{k+1} \) in Step 1 returns false then more work is needed to determine a verdict. We must decide whether the observed output \( o_{k+1} \) is some other counterexample to the correctness of \( \phi \) than \( p_{k+1} \). This situation easily occurs when the requirement \( \phi \) is a loose specification of the SUT behavior, such as a constraint or value interval. In this case we can evaluate the requirement formula \( \phi \) instantiated by the input and actual output sequences \( i_{k+1} \) and \( o_{k+1} \) to determine whether \( \phi \) is true or false. For this we perform a translation similar to \( \text{SatSet}_0(\phi) \) but with the variables \( x \) and \( out \) instantiated by the appropriate components of \( i_{k+1} \) and \( o_{k+1} \) respectively. We then evaluate all resulting sets of variable free equations and negated equations by rewriting. By Theorem 1, this approach will produce a correct verdict if \( o_{k+1} \) is a counterexample to \( \phi \).

Note that while Step 1 was already described in [16], Step 2 is an additional and more powerful step made possible by the translation of LTL into equational logic, and the use of term rewriting to implement the latter.

When the conditions of Theorem 3 are satisfied by the underlying data type then the LBT architecture of Figure 1 can be proven to terminate since the CGE algorithm correctly learns in the limit and basic narrowing is terminating and solution complete. A detailed analysis of this property will be published in an extended version of this paper. The argument is similar to that presented in [16].

6 A Case Study of LBT for Reactive Systems

Since the overhead of model checking an EMA by narrowing is high, it is important to study the performance of our LBT architecture in practice using case studies. Now many communication protocols can be modeled as EMA over (freely generated) symbolic data types. Thus the LTL decidability results of Section 4 apply to this class of examples.

The Transmission Control Protocol (TCP) is a widely used transport protocol over the Internet. We present here a performance evaluation of our LBT
architecture applied to testing a simplified model of the TCP/IP protocol as the
11 state EMA shown in Figure 2.

Fig. 2. TCP Mealy Machine Model

In this performance evaluation, we considered the fault detection capability of LBT compared with random testing. A coverage comparison of learning-based testing with random testing is for example [21], which even considers the same TCP case study. The methodology for comparison was to start from the concrete model of TCP in Figure 2 and consider a variety of correctness requirements as LTL formulas (including use cases). We then injected transition mutations into the SUT which falsified each individual requirement separately. In this way, several different kinds of bugs were introduced into the protocol model, such as mutating the input/output on transitions, adding extraneous transitions or states and so on. Some of these artificial mutations reflect realistic defects that have been discovered in several TCP/IP implementations [10].

Below we informally define five requirements on the TCP/IP protocol and give an LTL formalization of each.
1. Use case. Whenever the entity receives an active_open and sends out a SYN, the entity will send out a SYNACK if it receives a SYN, or send out an ACK if it receives a SYNACK, and send nothing when receiving other inputs.

\[ G((in = active\_open \land out = SYN) \rightarrow \]
\[ X((in = SYN \rightarrow out = SYNACK) \land (in = SYNACK \rightarrow out = ACK))) \]

2. Use case. Whenever the entity receives an active_open and sends out a SYN and then receives a SYNACK, the entity will send out an ACK and then will send out an ACK if it receives a FIN.

\[ G((in = active\_open \land out = SYN \land in = SYNACK) \rightarrow \]
\[ (X out = ACK \land X^2 (in = FIN \rightarrow out = ACK))) \]

3. Use case. Whenever the entity performs the IO (active_open, SYN) and receives SYNACK followed by FIN it will send out ACK followed by ACK and then send out FIN if it receives CLOSE.

\[ G((in = active\_open \land out = SYN \land X in = SYNACK \land X^2 in = FIN) \rightarrow \]
\[ (X out = ACK \land X^2 out = ACK \land X^3 (in = close \rightarrow out = FIN))) \]

4. Whenever the entity receives a close and sends out a FIN, or receives a FIN and sends out an ACK, the entity has either sent a passive_open or received an active_open before, and either sent or received a SYN before.

\[ G(((in = close \land out = FIN) \lor (in = FIN \land out = ACK)) \rightarrow \]
\[ (F^{-1} (in = pass\_open \lor in = active\_open) \land F^{-1} (in = SYN \lor out = SYN))) \]

5. Whenever the entity performs the IO (FINACK, ACK) it must have received or sent SYN in the past and performed the IO (close, FIN) in the past.

\[ G((in = FINACK \land out = ACK) \rightarrow \]
\[ (F^{-1} (in = SYN \lor out = SYN) \land F^{-1} (in = close \land out = FIN))) \]

### 6.1 Results and Analysis

To compare LBT with random testing on the TCP/IP stack model, we measured two related parameters, namely: (i) the time $t_{first}$ (in seconds), and (ii) the total number of queries (i.e. test cases) $Q_{first}$ needed to first discover an injected error in the SUT. To conduct random testing, we simply switched off the CGE and model checker algorithms. The performance of LBT is non-deterministic due to the presence of random queries. Therefore each value of $t_{first}$ and $Q_{first}$ is an average obtained from over 1000 LBT runs using the same injected error.
Table 1. Random testing versus LBT: a performance comparison

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Random Testing</th>
<th>LBT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$Q_{first}$ (sec)</td>
<td>$t_{first}$ (sec)</td>
</tr>
<tr>
<td>Req 1</td>
<td>101.4</td>
<td>0.11</td>
</tr>
<tr>
<td>Req 2</td>
<td>103.2</td>
<td>1.16</td>
</tr>
<tr>
<td>Req 3</td>
<td>1133.7</td>
<td>36.7</td>
</tr>
<tr>
<td>Req 4</td>
<td>582.82</td>
<td>1.54</td>
</tr>
<tr>
<td>Req 5</td>
<td>712.27</td>
<td>2.12</td>
</tr>
</tbody>
</table>

The results of testing the requirements Req1 to Req 5 are listed in Table 1. Note that $Q_{first}$ is the combined sum of the number of model checking queries $MCQ$ and random queries $RQ$. These are also listed in columns 6 and 7 to provide deeper insight into the strengths and weaknesses of our method. In the final column, $H_{hyp\_size}$ is the state space size of the learned hypothesis automaton at time $t_{first}$. Since $H_{hyp\_size}$ is always considerably less than 11 (the state space size of our SUT), this confirms the advantages of using an incremental learning algorithm such as CGE.

We wish to draw two main conclusions from Table 1.

(i) At the level of logical performance, (comparing $Q_{first}$ for LBT against $Q_{first}$ for random testing) we see that LBT always finds errors with significantly fewer test cases ranging between 0.25% and 18% of the number required by random testing. Therefore, if the overheads of model checking and learning can be reduced then LBT also has the potential to outperform random testing in real-time performance.

(ii) At the level of real-time performance (comparing $t_{first}$ for LBT against $t_{first}$ for random testing) we see that LBT is often but not always significantly faster than random testing, ranging between 0.9% and 160% of the time required by random testing. This reflects the actual real-time overhead of performing both model checking and learning for the SUT and each requirement.

Looking more closely at the results for Reqs 4 and 5, where LBT is somewhat slower than random testing, we can gain deeper insight into these real-time performance issues. For Reqs 4 and 5 both the values $MCQ$ and the ratios $RQ/MCQ$ are significantly higher than for Reqs 1, 2 and 3. In these cases, basic narrowing is performing a large number of constraint solving tasks on unsatisfiable sets of constraints. However, basic narrowing fails very slowly when no solutions can be found. After this, random test cases are applied to proceed with the task of learning the SUT, but these do not necessarily test the actual requirements.

These preliminary results are nevertheless promising, and based on them we make some suggestions for how to further improve narrowing in Section 7. Thus we can improve the overall real-time performance of our current LBT architecture to achieve a real-time performance closer to the logical performance. It should also be pointed out that as real-time measurement involves factors such
as efficiency of implementation, there exists further scope for improvement on the implementation level.

7 Conclusions

In this paper we have shown how a model checker based on narrowing can be combined with a symbolic automaton learning algorithm such as CGE to give a new architecture for black-box specification-based testing using the learning-based testing (LBT) paradigm. We have benchmarked this LBT architecture against random testing, and shown that it compares favorably, with the potential for future improvement.

The results of Section 6.1 suggest that a pure narrowing procedure could be significantly improved by interleaving it with theorem proving techniques to detect unsatisfiability. This is because counterexamples to correctness may be sparse, in which case narrowing fails very slowly. Term rewriting could be applied to this problem too. Furthermore, it is known that basic narrowing modulo theories is incomplete and suggestions such as the variant narrowing of [7] could be considered. Finally, we observe that the CGE algorithm does not currently learn infinite state Mealy automata, and this is another extension of our work that must be considered for EMA.

Acknowledgement

We gratefully acknowledge financial support for this research from the Chinese Scholarship Council (CSC), the Swedish Research Council (VR) and the European Union under project HATS FP7-231620. We also acknowledge the helpful comments of the referees in producing the final version of this report.

References