Case Studies in Learning-Based Testing

Lei Feng[1], Simon Lundmark[4], Karl Meinke[2], Fei Niu[2],
Muddassar A. Sindhu[5], and Peter Y.H. Wong[3]

[1] Machine Design Department,
Royal Institute of Technology, Stockholm 10044, Sweden
feng@kth.se
[2] School of Computer Science and Communication,
Royal Institute of Technology, Stockholm 10044, Sweden
{karlm,niu}@csc.kth.se
[3] SDL Fredhopper, Amsterdam, The Netherlands
peter.wong@fredhopper.com
[4] TriOptima AB, Stockholm, Sweden
simon.lundmark@trioptima.com
[5] Computer Science Department, Quaid i Azam University, Islamabad, Pakistan
masindhu@qau.edu.pk

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

1 Introduction

Learning-based testing (LBT) [7] is an emerging paradigm for *black-box requirements testing* that automates the three basic steps of: (1) automated test case generation (ATCG), (2) test execution, and (3) test verdict (the oracle step).

The basic idea of LBT is to automatically generate a large number of high-quality test cases by combining a model checking algorithm with an *incremental model inference* or *learning algorithm.* These two algorithms are integrated with the *system under test* (SUT) in an iterative feedback loop. On each iteration of this loop, a new test case can be generated either by: (i) model checking a learned model M_i of the system under test (SUT) against a formal user requirement *req* and choosing any counterexample to correctness, (ii) using the learning algorithm to generate a membership query, or (iii) random generation. An LBT tool must *interleave* these three TCG methods to achieve an overall testing strategy that is efficient. Whichever TCG method is chosen, the new test case t_i is then executed on the SUT, and the outcome is judged as a pass, fail or warning. This is done by comparing a predicted output p_i (obtained from M_i) with the observed output o_i (from the SUT). The new input/output pair (t_i, o_i) is also used to update the

H. Yenigün, C. Yilmaz, and A. Ulrich (Eds.): ICTSS 2013, LNCS 8254, pp. 164–179, 2013.
© IFIP International Federation for Information Processing 2013

current model M_i to a refined model M_{i+1}, which ensures that the iteration can proceed again. If the learning algorithm can be guaranteed to correctly learn in the limit, given enough information about the SUT, then LBT is a sound and complete method of testing. In practice, real-world systems are often too large for complete learning to be accomplished within a feasible timescale. By using incremental learning algorithms, that focus on learning just that part of the SUT which is relevant to the requirement *req*, LBT becomes much more effective.

While algorithms for LBT have been analyzed and benchmarked on small academic case studies (see [9] and [12]), there has so far been no published evaluation of this technology on real-world case studies. So the scalability of this approach is unclear. The work presented here therefore has two aims:

1. to describe the problems and potential of using LBT on real world systems from a variety of industrial domains;
2. to show that learning-based testing is scalable to large industrial case studies, by measuring concrete performance parameters of the LBT tool *LBTest*.

The organization of this paper is as follows: In Section 3, we give an introduction to requirements testing with the *LBTest* tool focussing on its requirements language. In Section 4, we describe industrial case studies with the *LBTest* tool from three different industrial domains: *web, automotive* and *finance*. Finally in Section 5, we give some conclusions and future directions of research.

2 Related Work

A tutorial on the basic principles of LBT and their application to different types of SUTs can be found in [10]. The origin of some of these ideas can be traced perhaps as far back as [17]. Experimental studies of LBT using different learning and model checking algorithms include [12], [7], [8] and [9]. These experiments support the thesis that LBT can substantially outperform random testing as a black-box requirements testing method.

Several previous works, (for example Peled et al. [13], Groce et al. [5] and Raffelt et al. [14]) have also considered a combination of learning and model checking to achieve testing and/or formal verification of reactive systems. Within the model checking community, the verification approach known as *counterexample guided abstraction refinement* (CEGAR) also combines learning and model checking (see e.g. Clarke et al. [3] and Chauhan et al. [1]). The LBT approach can be distinguished from these other approaches by: (i) an emphasis on testing rather than verification, and (ii) use of *incremental learning algorithms*, as well as other optimisations, specifically chosen to make testing more effective and scalable. This related research does not yet seem to have lead to practical testing tools. *LBTest* is the first LBT tool to be used in industrial environments.

Inductive testing (Walkinshaw et al. [16]) is a black-box testing technique that also uses automata learning algorithms. However, this approach is more focussed on finding untrapped exceptions then testing formal user requirements (model checking is not used). Despite its different aim, [16] confirms our own findings that learning algorithms give more efficient search methods than random testing.

In contrast with *model-based testing tools*, such as *Conformiq Designer* [4] or *ModelJUnit* [15], which perform test case generation using a design model (such as a UML model), *LBTest* reverse engineers its own models for testing purposes. Thus *LBTest* is advantageous in agile development since its models do not have to be manually designed or re-synchronised with code changes.

3 Requirements Testing with *LBTest*

A platform for learning-based testing known as *LBTest* [11] has been developed within the EU project HATS FP7-231620. This platform supports black-box requirements testing of fairly general types of reactive systems. The main constraint on applying *LBTest* is that it must be possible to model a particular SUT by a deterministic finite state machine.

The inputs to *LBTest* are a black-box SUT and a set of formal user requirements to be tested. The tool is capable of generating, executing and judging a large number of tests cases within a short time. In large case studies, the main limitation on test throughput is the average execution time of a single test case on the SUT. (This will be seen in case study 3 of Section 4.3.)

For user requirements modeling, the formal language currently supported in *LBTest* is *propositional linear temporal logic* (PLTL) extended by *finite data types*. PLTL formulas can express both: (i) *safety properties* which are invariants that may not be violated, and (ii) *liveness properties*, including *use cases*, which specify intended dynamic behaviors. A significant contribution of *LBTest* is its support for liveness testing. Our case studies in Section 4 will provide examples of both safety and liveness testing.

Currently in *LBTest*, only one (external) model checker is supported, which is NuSMV [2]. Further interfaces are planned in the future. The learning algorithm currently available in *LBTest* is the IKL algorithm [12], which is an algorithm for incremental learning of deterministic Kripke structures. New learning algorithms are also in development for future evaluation.

3.1 PLTL as a Requirements Modeling Language

In the context of reactive systems analysis, temporal logics have been widely used to formally model user requirements. From a testing perspective, *linear time temporal logic* (LTL) with its emphasis on the properties of paths or execution sequences, is a natural choice. The design philosophy of *LBTest* is to generate, execute and judge as many test cases as possible within a given time limit. This requirement places stringent requirements on the efficiency of model checking LTL formulas. Therefore, only model checking of *propositional linear temporal logic* (PLTL) formulas is currently considered. However, in an effort to make PLTL more user-friendly (by hiding low-level Boolean encodings) *LBTest* supports an extended PLTL with user-defined symbolic finite data types.

To use *LBTest* correctly it is important to understand the precise syntax of the requirements modeling language. Our data type model is based on the well

known algebraic model of *abstract data types*, involving many-sorted signatures and algebras (see e.g. [6]).

3.1.1 Definition

A *finite data type signature* Σ consists of a finite set S of *sorts* or types, and for each sort $s \in S$, a finite set Σ_s of *constant symbols* all of the same type s.

3.1.2 Definition

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

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

where $s \in S$ and $c \in \Sigma_s$. This logic has a simple but strict typing system.

The atomic formulas of $PLTL(\Sigma)$ are equations and inequations over the data type signature Σ for defining input and output operations. Only a single variable symbol of each type is allowed, to support a *simple black-box interface* to the SUT. We overload each type symbol $s \in S$ to also name a unique SUT read or write variable of type s. The distinguished sort $in \in S$ denotes the single *SUT write variable*, while every other type $s \in S$ denotes an *SUT read variable.*

The language $PLTL(\Sigma)$ can be given a formal Kripke semantics, in a routine way, over any algebra that interprets the data type signature Σ. A precise definition is omitted for brevity. Informally, the symbols $\bot$, $\top$, $\neg$, $\wedge$, $\vee$ and $\rightarrow$ denote the usual Boolean constants and connectives. The symbols X, F, G, U, W and R denote the temporal operators. Thus, $X\phi$ means that ϕ is true in the next state, $F\phi$ means that ϕ is true sometime in the future, $G\phi$ means that ϕ is always true in the future and U is the binary operator which means that ϕ_1 will remain true until a point in the future when ϕ_2 becomes true. The two operators W and R stand for *weak until* and *release* respectively.

4 Case Studies in Learning-Based Testing

We can now present three industrial case studies which were tested with *LBTest*. These were: (i) an *access server* (FAS) from Fredhopper , (ii) a *break-by-wire system* (BBW) from Volvo Technology, and (iii) a *portfolio compression service* (triReduce) from TriOptima . These case studies represent mature applications from the domains of web, automotive and finance. The tool was able to find errors in each of them, which is a promising achievement. These case studies have the following basic characteristics:

- FAS is an e-commerce application which has been developed and evolved over 12 years. Its various modules have been tested with automated and manual techniques. Requirements modeling involved events and finite data types.

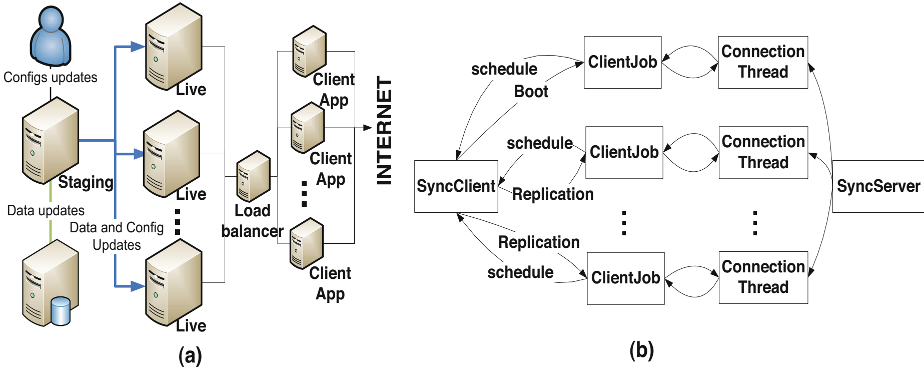

Fig. 1. (a) an FAS Deployment and (b) Interactions in the Replication System

– BBW is relatively new embedded application from the automobile industry, which has not yet been widely adopted. It has strict timing constraints to ensure the safety of the vehicle. Requirements modeling involved events and infinite data types.
– triReduce service has been developed using Django, a popular web framework for the Python programming language. It involves significant use of dynamically changing databases, so test set-up and tear-down are non-trivial. Requirements modeling involved events and finite data types.

Requirements modeling for the BBW case study was particularly challenging due to the presence of the infinite floating point data type. This required an extension of *LBTest* to support *partition testing* by *discretising* each floating point domain.

4.1 Case Study 1: Access Server

The Fredhopper Access Server (FAS) is a distributed, concurrent OO system developed by Fredhopper that provides search and merchandising services to e-Commerce companies, including structured search capabilities within the client's data. Fig. 1(a) shows the deployment architecture used to deploy an FAS to a customer. An FAS consists of a set of live environments and a single staging environment. A live environment processes queries from client web applications via web services. A staging environment is responsible for receiving data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments according to the Replication Protocol. The Replication Protocol is implemented by the Replication System which consists of a SyncServer at the staging environment and one SyncClient for each live environment. The SyncServer determines the schedule of replication jobs, as well as their contents, while SyncClient receives data and configuration updates according to the schedule. Fig. 1(b) shows the interactions in the Replication System. Informally, the

Replication Protocol is as follows: the SyncServer begins by listening for connections from SyncClients. A SyncClient creates and schedules a ClientJob object with job type Boot that connects immediately to the SyncServer. The SyncServer then creates a ConnectionThread to communicate with the SyncClient's ClientJob. The ClientJob asks the ConnectionThread for replication schedules, notifies the SyncClient about the schedules, receives a sequence of file updates according to the schedule from the ConnectionThread and terminates.

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

The SUT was a Java implementation of the SyncClient, consisting of about 6400 lines of Java code organized into 44 classes and 2 interfaces. Specifically, we were interested to test the interaction between a SyncClient and a ClientJob by learning the SyncClient as a Kripke structure over the input data type

$$\Sigma_{in} = \{setAcceptor,\ schedule,\ searchjob,\ businessJob,\ dataJob,$$
$$connectThread,\ noConnectionThread\}$$

Four relevant output data types were identified as follows:

$$\Sigma_{schedules} = \{\phi, \{search\}, \{business\}, \{business,\ search\}, \{data\},$$
$$\{data,\ search\}, \{data,\ business\}, \{data,\ business,\ search\}\}.$$
$$\Sigma_{state} = \{Start, WaitToBoot, Boot, WaitToReplicate, WorkOnReplicate,$$
$$WorkOnReplicate, End\},$$
$$\Sigma_{jobtype} = \{nojob, Boot, SR, BR, DR\},\quad \Sigma_{files} = \{readonly, writeable\}.$$

Eleven informal user requirements were then formalized in $PLTL(\Sigma)$. Below, for brevity, we only reproduce some of these requirements formally.

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

$$G(state\ = Start \wedge in = setAcceptor \rightarrow X(state\ =\ WaitToBoot \wedge jobtype = Boot))$$

Requirement 2: *If the SyncClient's state is either WaitToBoot or Booting then it must have a boot job (Jobtype = Boot), and if it has a boot job, its state can only be one of WaitToBoot, Booting, WaitToReplicate or End.*[1]

$$G(state \in \{WaitToBoot,\ Booting\} \rightarrow jobtype\ =\ Boot \wedge$$

[1] The membership relation $\in$ used in Requirement 2 and elsewhere does not belong to PLTL(Σ) but is a macro notation that can be replaced automatically.

$jobtype = Boot \rightarrow state \in \{WaitToBoot, Booting, WaitToReplicate, End\})$

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

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

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

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

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

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

$$G(state \neq End \rightarrow$$

$$search \in schedules \rightarrow (F(jobtype = SR\ \ U\ state = End)) \wedge$$

$$business \in schedules \rightarrow (F(jobtype = BR\ \ U\ state = End)) \wedge$$

$$data \in schedules \rightarrow (F(jobtype = DR\ \ U\ state = End))\ )$$

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

Requirement 10: *If the SyncClient is executing a replication job for a particular type of schedule, then that job can only receive schedules for that particular type of schedule.*

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

Table 1 gives the results obtained by running *LBTest* to test these 11 user requirements on the FAS SyncClient. For each requirement, Table 1 breaks down the *total number* of test cases used into three figures (columns 5, 6 and 7) which count the test cases generated by each of the three different TCG methods: *model checker*, *learner* and *random*. The total testing time (column 3) is the time taken to execute all three types of test cases. For each requirement, Table 1 gives the final verdict (column 2) i.e. *pass/fail/warning*. Column 4 gives the size of the learned hypothesis model at test termination. To terminate each experiment, a maximum time bound of 5 hours was chosen. However, if the hypothesis model size had not changed over 10 consecutive random tests, then testing was terminated earlier than this.

Table 1. Performance of *LBTest* on Fredhopper Access Server case study

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

Thus for example: Requirement 1 was tested for a total of 5 hours using 50,942 test cases, of which 50,897 were generated by the learning algorithm, 45 were generated randomly, and 0 were generated by the model checker. We see that learner generated queries dominate, though generally this is influenced by the kind of learning algorithm used (here IKL [12]). Around 10,000 test cases per hour were generated, executed and evaluated. This test throughput does not vary much across the 11 different requirements. On large SUTs, test throughput is mainly determined by the average execution speed of a single test case. Since Requirement 1 was passed, we can infer that the model checker was called 45 times, but on each occasion it failed to find a counterexample, so that a random test case was used instead.

4.1.1 Discussion of Errors Found

Nine out of eleven requirements were passed. For Requirements 8 and 9, *LBTest* gave warnings (due to a loop in the counterexample) corresponding to tests of liveness requirements that were never passed. The counterexample for both these requirements was "*setAcceptor,Schedule,businessJob,businessJob*". After the first instance of symbol *"businessJob"* , a loop occurred in the counterexample which was unfolded just once. This counterexample violated Requirement 8 because if we keep reading the input *businessJob* from the state reached after the first *"businessJob"* the SUT does not go to the end state as specified. It also violates Requirement 9 because the start state is reached after reading this sequence rather than *WaitOnReplicate* or *End* states as specified. Neither of these states is ever reached if we keep reading the input *businessJob* from this state. A careful analysis of these requirements showed that both involved using the U (strong Until) operator. When this was replaced with a W (weak Until) operator no further warnings were seen for Requirement 9. Therefore this was regarded as an

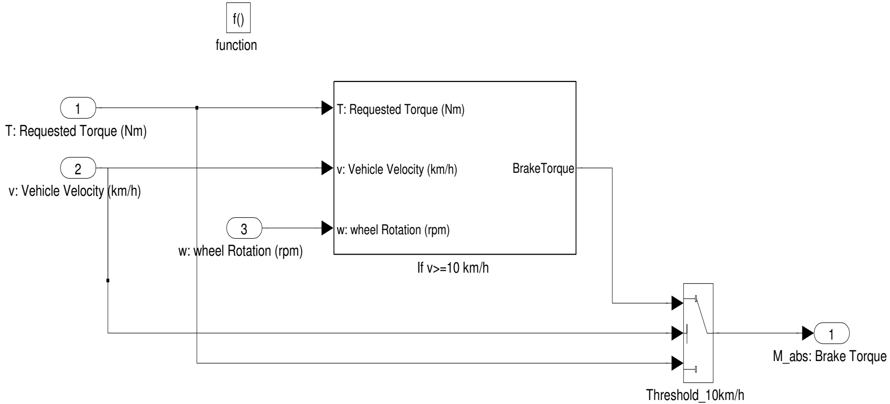

Fig. 2. A High Level Simulink Model of the BBW System

error in the user requirements. However, *LBTest* continued to produce warnings for Requirement 8, corresponding to a true SUT error. So in this case study *LBTest* functioned to uncover errors both in the user requirements and in the SUT.

4.2 Case Study 2: Brake-by-Wire

The Volvo Technology BBW system is an embedded vehicle application with ABS function, where no mechanical connection exists between the brake pedal and the brake actuators applied to the four wheels. A sensor attached to the brake pedal reads the pedal's position percentage, which is used to compute the desired global brake torque. A software component distributes this global brake torque request to the four wheels. At each wheel, the ABS algorithm uses the corresponding brake torque request, the measured wheel speed, and the estimated vehicle speed to compute the actual brake torque on the wheel. For safety purposes, the ABS controller in the BBW system must release the corresponding brake actuator when the slip rate of any wheel is larger than the threshold (e.g., 20%) and the vehicle is moving at a speed of above certain value, e.g., 10 km/h. A Simulink model of the BBW system is shown in Figure 2.

The BBW is a typical distributed system, which is realised by five ECUs (electronic control units) connected via a network bus. The central ECU is connected to the brake and acceleration (gas) pedals. The other four ECUs are connected to four wheels. The software components on the central ECU manage the brake pedal sensor, calculation of the global brake torque from the brake pedal position, and distribution of the global torque to the four wheels. The software components on each wheel ECU measure the wheel speed, control the brake actuator, and implement the ABS controller. The BBW is a hard real-time system with some concrete temporal constraints that could not be modeled in PLTL.

For this it runs 'continuously' with high frequency sampling using two clocks (5 and 20 ms). The BBW has:

- *two real-valued inputs*: received from the brake and gas pedals to identify their positions, denoted by *breakPedalPos* and *gasPedalPos* respectively. The positions of both pedals are bounded by the interval $[0.0, 100.0]$.
- *nine real-valued outputs*: denoting the vehicle speed *vehSpeed*, rotational speeds of the four wheels (front right, front left, rear right and rear left) $\omega SpeedFR$, $\omega SpeedFL$, $\omega SpeedRR$ and $\omega SpeedRL$ respectively. These speeds are bounded by the interval $[0.0, 111.0]$. The torque values on these wheels are denoted by *torqueOnFR, torqureOnRL, torqueOnRR* and *torqueOnRL* respectively. All torque values are bounded by the interval $[0.0, 3000.0]$ Nm.

The SUT consisted of a Java implementation of the BBW consisting of about 1100 lines of code.

The problem of discretely modeling floating point data types was addressed by *partition testing*. For this, the two inputs were discretised into a set of four input events given by $\Sigma_{in} = \{brake, acc, accbrake, none\}$, where the values *brake* and *acc* represented the conditions $brakePedalPos = 100.0$ and $accPedalPos = 100.0$ respectively. Also *accbrake* represented the condition $brakePedalPos = 100.0 \wedge accPedalPos = 100.0$ and *none* represented $brakePedalPos = 0.0 \wedge accPedalPos = 0.0$ respectively.

Exploiting symmetrical user requirements on all four wheels, four finite output data types for the vehicle speed and one single wheel (front right) were identified as:

$$\Sigma_{speed} = \{vehicleStill, vehicleMove, vehicleDecreased\}$$

$$\Sigma_{wheelRotateFR} = \{zero, nonZero\}$$

$$\Sigma_{torqueFR} = \{zero, nonZero\}$$

$$\Sigma_{slipFR} = \{slipping, notSlipping\}$$

The floating point values of each output variable were mapped to the corresponding set of output events in Σ by using *discretisation formulas*. Effectively, each event represents a partition class. These formulas are defined in Table 2, and they were implemented within the SUT adapter which was used to connect the BBW to *LBTest*.

Note that in Table 2, $vehSpeed_i$ represents the vehicle speed at *i-th* event and hence the speed change at *i-th* event is $vehSpeed_i - vehSpeed_{i-1}$. The units of measurement for $vehSpeed_i$ are km/h and the vehicle is considered as still at the *i-th* event if $vehSpeed_i \leq 10$ otherwise it is considered to be in motion. The vehicle is considered to be decelerating at the i-th event if $vehSpeed_i < vehSpeed_{i-1}$ and $vehSpeed_{i-1} > 0$. The units of angular speed of the wheels are also converted into km/h inside the Java code of the SUT from the usual rpm. This was essential to calculate the slip rate of the wheels. The slip rate (denoted by *slip*) of a wheel (e.g front right) is the ratio of the difference of $vehSpeed - \omega SpeedFR$ and the vehicle speed *vehSpeed*. The vehicle is considered slipping when the slip rate $slip > 0.2$ otherwise the vehicle is considered not

Table 2. BBW Discretisation Formulas

Output Value	Discretisation Formula
vehicleStill	$vehSpeed \leq 10.0$
vehicleMove	$vehSpeed > 10.0$
vehicleDecreased	$vehSpeed_i < vehSpeed_{i-1} \wedge vehSpeed_{i-1} > 0$
nonZero : wheelRotateFR	$\omega SpeedFR > 0$
Zero : wheelRotateFR	$\omega SpeedFR = 0$
nonZero : torqueFR	$torqueOnFR > 0$
Zero : torqueFR	$torqueOnFR = 0$
slipping	$10 * (vehSpeed - \omega SpeedFR) > 2 * vehSpeed$

slipping. After this partitioning of the input and output values, three informal requirements were formalized in $PLTL(\Sigma)$ as follows:

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

$$G(\, in = brake \rightarrow F(wheelRotateFR = nonZero \rightarrow torqueFR = nonZero)\,)$$

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

$$G(\,(in = brake \;\wedge\; speed = vehicleMove \;\wedge\; slipFR = slipping) \rightarrow torqueFR = zero\,)$$

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

$$G(in = accbrake \rightarrow X(speed = vehicleDecreased))$$

Table 3. Performance of *LBTest* on the Brake-by-Wire case study

PLTL Requirement	Verdict	Total Testing Time (min)	Hypothesis Size (States)	Model Checker Tests	Learning Tests	Random Tests
Req 1	Pass	34.4	11	0	1501038	150
Req 2	Fail	1.0	537	18	34737	2
Req 3	Pass	16.0	22	0	1006275	130

4.2.1 Discussion of Errors Found

Table 3 shows the results of testing BBW with *LBTest* using the three LTL requirements defined above. Noteworthy are the large volumes of test cases and short session times, due to fast execution of individual test cases. Requirements 1 and 3 were passed, while *LBTest* continued to give errors for Requirement 2 with different counterexamples during several testing sessions we ran. The shortest counterexample found during these sessions was "*acc, acc, acc, acc, acc, brake*". This means that when the brake pedal is pressed, after the vehicle has acquired a speed greater than 10 km/h, and at that time when the slip rate of a wheel is greater than 20%, then the SUT does not always have zero torque on the slipping wheel. All other counterexamples suggested a similar pattern of behaviour.

4.3 Case Study 3: triReduce

TriOptima is a Swedish IT company in the financial sector which provides post-trade infrastructure and risk management services for the over-the-counter (OTC) derivatives market. Financial markets involve a constantly changing and strict regulatory framework. To keep up with such changes, agile software development methodologies are important. However, short development cycles are difficult without test automation, and ATCG is therefore a valuable addition to quality assurance methods.

A *derivative* is a financial instrument whose value derives from the values of other underlying variables. It can be used to manage financial risk. Millions of derivatives are traded every day. Yet many of these trades are not necessary to maintain a desired risk position versus the market. Financial institutions can participate in *portfolio compression* activities to get rid of unnecessary trades. The triReduce portfolio compression service runs in *cycles*, each cycle focuses on a different product type, for example interest rate swaps or credit default swaps. Each cycle has four main steps:

1. *Preparation.* Before participating, each financial institution must complete a legal process which results in a *protocol adherence.*
2. *Sign up.* Parties can log in to the service to review the schedule for upcoming compression cycles and sign up, indicating they will participate.
3. *Linking.* During this phase, each participating party uploads their portfolio of trades, complementing the other participants in the cycle. The trades are automatically matched, determining which trades are eligible for compression. During linking, each participant sets their parameters for controlling movements in the market and credit risk.
4. *Live execution.* After calculating a multilateral unwind proposal, the different parties verify this and indicate acceptance. When all participants have indicated acceptance, the proposed trades are legally terminated.

TriOptima delivers the triReduce service through a web application that is developed in Python using the Django Framework. Django is based on a Model-View-Controller-like (MVC) software architecture pattern, where the *models* are tightly coupled to relational database models, implemented as Python classes.

In general, testing TriOptima involves *isolating* test case executions so that the ordering of test cases in a suite does not matter. Clearing the databases and caches between test executions would properly isolate them. However, setting up and tearing down the entire database is not feasible in this case since triReduce has a code base almost 100 times larger than the FAS at about 619 000 LoC (including large dependencies like Django). It is a database intensive application with customers uploading trade portfolios containing thousands of trades. Handling this data efficiently takes several hundred database models and relational database tables. Creating these blank database tables and their indexes can take time in the order of minutes. Therefore, a special SUT adapter was written to effectively isolate each test case execution by performing an efficient database rollback between individual test cases. This SUT adapter was implemented in a general way that could be used for any other Django web application.

Authentication solves the problem of deciding *who* a certain user is. Authorization solves the problem of deciding if a certain (authenticated) user is *allowed* to perform a certain operation. A central component of any Django application is its user authentication system. A central and extremely important part of triReduce is deciding what a user may or may not do in the system. For example, a simplified view of the user authorization in triReduce can be described as follows. A *user* is a member of a *party* (a legal entity). There are also user accounts for TriOptima *staff*. To sign up for a triReduce compression cycle, the party must have the correct *protocol adherence* stored in the system. (The protocol can be seen as a legally binding contract between TriReduce and the party.) Only staff users may add or remove protocol adherences for parties. Because of their critical importance for system integrity, requirements related to authentication and authorization were the focus of this *LBTest* case study.

For this specific focus, the input data type was defined by:

$$\Sigma_{in} = \{next_subject, login, adhere_to_protocol\}$$

and four relevant output data types were identified as:

$$\Sigma_{status} = \{ok, client_error, internal_error\}$$

$$\Sigma_{subject} = \{none, root, a_alice, b_bob\}$$

$$\Sigma_{logged_in} = \{anonymous, staff, bank_a, bank_b\}$$

$$\Sigma_{protocol} = \{not_adheres, adheres\}, \quad \Sigma_{signup} = \{prohibited, allowed\}.$$

Five functional requirements were then formalised in $PLTL(\Sigma)$:

Requirement 1: *The status must always be okay.*

$$G(status = ok)$$

Requirement 2: *If Bank A is not logged in, and does log in, then Bank A should become logged in.*

$$G(logged_in \in \{bank_b,\ staff,\ anonymous\} \wedge$$
$$subject = a_alice\ \wedge\ in = login \rightarrow X(logged_in = bank_a))$$

Requirement 3: *Cycle signup should be prohibited until a bank adheres to the protocol.*

$$G((logged_in = bank_a \rightarrow cycle_signup = prohibited)\ U$$

$$(logged_in = bank_a \rightarrow adheres_to_protocol = adheres))$$

Requirement 4: *Cycle signup should be prohibited until a bank adheres to the protocol, and general system status should always be ok, i.e. 3 and 1 together.*

Requirement 5: *If bank A adheres to the protocol, then cycle signup for bank A should always be allowed.*

$$G((logged_in = bank_a \rightarrow adheres_to_protocol = adheres) \rightarrow$$

$$G(logged_in = bank_a \rightarrow cycle_signup = allowed))$$

4.3.1 Discussion of Errors Found

Three different types of errors were found within four types of experiments, which could be classified as follows.

1. Injected Errors. Injecting errors into the SUT was a way of confirming that *LBTest* was working as intended. Below, we describe some examples of doing this, and the results may be seen in Table 4. Here, three versions of triReduce were used:

- triReduce 1: The standard version.
- triReduce 2: An error is injected, the password of the `b_bob` user was changed.
- triReduce 3: Another error is introduced, switching the meaning of logging in as user `a_alice` and `b_bob`.

2. Errors in the SUT Adapter. While testing Requirement 3 and observing the log output, some SUT output contained `status` states that were not `ok`. Thus internal errors were arising in the SUT, which no requirement had covered. Therefore Requirement 1 and Requirement 3 were combined resulting in Requirement 4. After about 55 minutes of *LBTest* execution it found a counterexample to Requirement 4. This error was easily traced to the SUT adapter, the code connecting *LBTest* to triReduce, and was quickly fixed.

Table 4. Results of injecting errors into triReduce

Req. #	SUT	Verdict	Comment
1	triReduce 1	Pass	Stopped after learning a model of 16 states using 5 hypothesis models, after 18 min.
1	triReduce 2	Warning	Counterexample found in the sixth hypothesis (size 8 states) after only 3.8 min.
2	triReduce 1	Pass	As previously, stopped after learning 5 hypothesis models in 13 min.
2	triReduce 3	Warning	Counterexample found after 98 seconds at a hypothesis size of 4 states.

3. Errors in Requirements. Executions of *LBTest* found an error in the original formalisation of Requirement 3, due to using $\wedge$ instead of $\rightarrow$ (a common mistake for beginners). *LBTest* was able to detect this, by producing a spurious counterexample within a minute on the faulty LTL requirement.

4. Successful Lengthy LBTest Executions. Requirement 5, was tested to see how *LBTest* would behave in much longer testing sessions. Two 7 hour testing sessions were successfully executed with *LBTest*. Both terminated with a "pass" verdict after about 86000 SUT executions and hypothesis sizes of up to 503 states. The log files were manually checked and contained no errors.

5 Conclusions and Future Work

We have applied *LBTest*, a learning-based testing tool, to three industrial case studies from the web, automotive and finance sectors. The tool successfully found errors in all three case studies (albeit injected errors for triReduce). This is despite the fact that one case study (the FAS) had been operational for a relatively long time. The tool supported formal requirements debugging in two case studies, which is often considered to be problematic. The successes of these large case studies suggest that LBT is already a scalable technique, that could be further improved with better learning algorithms.

The third case study (triReduce), is the largest that has been tested using *LBTest* to date. While no SUT errors were found, we note that this study was performed by a test engineer external to the original *LBTest* research team. An early version of *LBTest*, with limited documentation and guidance was used. This suggests that LBT technology should be transferable to industry.

These case studies illustrate the scope and potential for LBT within different industrial domains and problems. They also illustrate the practical difficulties of using LBT within an industrial environment, including requirements modeling and implementing appropriate SUT adapters.

Future research will consider more efficient learning algorithms which can reduce both the number of test cases and the time needed to discover errors. The combination of partition testing with LBT, used in the BBW case study, also merits further research to understand its scope and limits. In [9], we investigated extensions of LBT with more powerful model checkers for full first-order linear temporal logic. However, it remains to be seen whether this approach is competitive with the much simpler but less precise partition testing method used here.

We gratefully acknowledge financial support for this research from the Higher Education Commission (HEC) of Pakistan, the Swedish Research Council (VR) and the European Union under project HATS FP7-231620 and ARTEMIS project 269335 MBAT.

References

1. Chauhan, P., Clarke, E.M., Kukula, J.H., Sapra, S., Veith, H., Wang, D.: Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 33–51. Springer, Heidelberg (2002)
2. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model verifier. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 495–499. Springer, Heidelberg (1999)
3. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
4. conformiq. The conformiq designer tool, `http://www.conformiq.com/products/conformiq-designer/`
5. Groce, A., Peled, D., Yannakakis, M.: Adaptive model checking. Logic Journal of the IGPL 14(5), 729–744 (2006)
6. Loeckx, J., Ehrich, H.-D., Wolf, M.: Specification of abstract data types. Wiley (1996)
7. Meinke, K.: Automated black-box testing of functional correctness using function approximation. In: ISSTA 2004: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 143–153. ACM, New York (2004)
8. Meinke, K., Niu, F.: A learning-based approach to unit testing of numerical software. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 221–235. Springer, Heidelberg (2010)
9. Meinke, K., Niu, F.: Learning-based testing for reactive systems using term rewriting technology. In: Wolff, B., Zaïdi, F. (eds.) ICTSS 2011. LNCS, vol. 7019, pp. 97–114. Springer, Heidelberg (2011)
10. Meinke, K., Niu, F., Sindhu, M.: Learning-based software testing: a tutorial. In: Hähnle, R., Knoop, J., Margaria, T., Schreiner, D., Steffen, B. (eds.) ISoLA 2011 Workshops 2011. CCIS, vol. 336, pp. 200–219. Springer, Heidelberg (2012)
11. Meinke, K.: Muddassar A. Sindhu. LBTest: A learning-based testing tool for reactive systems. In: Proc. of the Sixth IEEE Int. Conf. on Software Testing, Verification and Validation, ICST 2013. IEEE Computer Society (to appear, 2013)
12. Meinke, K., Sindhu, M.A.: Incremental learning-based testing for reactive systems. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 134–151. Springer, Heidelberg (2011)
13. Peled, D., Vardi, M.Y., Yannakakis, M.: Black-box checking. In: Formal Methods for Protocol Engineering and Distributed Systems FORTE/PSTV, pp. 225–240. Kluwer (1999)
14. Raffelt, H., Steffen, B., Margaria, T.: Dynamic testing via automata learning. In: Yorav, K. (ed.) HVC 2007. LNCS, vol. 4899, pp. 136–152. Springer, Heidelberg (2008)
15. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco (2006)
16. Walkinshaw, N., Bogdanov, K., Derrick, J., Paris, J.: Increasing functional coverage by inductive testing: a case study. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 126–141. Springer, Heidelberg (2010)
17. Weyuker, E.: Assessing test data adequacy through program inference. ACM Trans. Program. Lang. Syst. 5(4), 641–655 (1983)