# Case Studies in Learning-based Testing

Lei Feng
Machine Design Department
Royal Institute of Technology
Stockholm, Sweden
feng@kth.se

Karl Meinke
School of Computer Science
and Communication (CSC)
Royal Institute of Technology
Stockholm 10044, Sweden
karlm@csc.kth.se

Fei Niu
School of Computer Science
and Communication (CSC)
Royal Institute of Technology
Stockholm 10044, Sweden
niu@csc.kth.se

Muddassar A. Sindhu
School of Computer Science
and Communication (CSC)
Royal Institute of Technology
Stockholm 10044, Sweden
sindhu@csc.kth.se

Peter Y. H. Wong
SDL Fredhopper, Amsterdam
The Netherlands
peter.wong@fredhopper.com

## ABSTRACT

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

## 1. INTRODUCTION

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

The basic idea of LBT is to automatically generate a large number of high-quality test cases by combining a model checking algorithm with an *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. Whichever method is chosen, the new test case $t_i$ is then executed on the SUT, and the outcome is judged as a pass, fail or warning. This is done by comparing a predicted output $p_i$ (obtained from $M_i$) with the observed output $o_i$ (from the SUT). The new input/output pair $(t_i, o_i)$ is also used to update the current model $M_i$ to a refined model $M_{i+1}$, which ensures that the iteration can proceed again. If the learning algorithm can be guaranteed to correctly learn in the limit, given enough information about the SUT, then LBT is a sound and complete method of testing. In practice, real-world systems are often too large for complete learning to be accomplished within a feasible timescale. By using incremental learning algorithms, that can focus on learning just that part of the SUT which is relevant to the requirement *req*, LBT becomes much more effective.

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

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

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

## 2. LITERATURE AND TOOL SURVEY

A tutorial on the basic principles of LBT and their application to different types of SUTs can be found in [15]. The origin of some of these ideas can be traced perhaps as far

back as [22]. Experimental studies of LBT using different learning and model checking algorithms include [17], [12], [13] and [14]. These experiments have repeatedly shown that LBT can substantially outperform random testing as a black-box testing method.

Several previous works, (for example Peled et al. [18], Groce et al. [10] and Raffelt et al. [19]) have also considered a combination of learning and model checking to achieve testing and/or formal verification of reactive systems. Within the model checking community the verification approach known as *counterexample guided abstraction refinement* (CEGAR) also combines learning and model checking, (see e.g. Clarke et al. [7] and Chauhan et al. [5]). The LBT approach can be distinguished from these other approaches by: (i) an emphasis on testing rather than verification, and (ii) use of *incremental learning algorithms* specifically chosen to make testing more effective and scalable. This related research does not yet seem to have lead to practical testing tools. So LBTest is the first experimental testing tool to be made available that combines automata learning methods with model checker based TCG.

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

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

# 3. REQUIREMENTS TESTING WITH LBTEST

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

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

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

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

## 3.1 PLTL as a Requirements Modeling Language

In the context of reactive systems analysis, temporal logics have been widely used to model user requirements. Indeed, semi-formal user requirements, languages such as UML sequence diagrams, can be given a precise semantics in terms of temporal logic.

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

To use the LBTest tool correctly it is important to understand the precise syntax of PLTL which is supported. Furthermore, to understand the case studies of Section 3, and to conduct new case studies, it is important to have at least an informal understanding of PLTL semantics. Therefore we shall define these two aspects of the LBTest language interface in this section.

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

types by finite ones.

### 3.1.1 Definition

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

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

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

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

### 3.1.2 Syntax of $PLTL(\Sigma)$

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

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

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

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

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

## 3.2 Finite Data Type Modeling and Interface Definitions

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

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

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

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

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

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

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

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

| Constant Symbol | Encoding |
|---|---|
| *brake* | *a* |
| *dec* | *b* |
| *gas* | *c* |
| *acc* | *d* |
| *button* | *e* |

**Table 1: Input Data Type for CC**

| Data Type | Start index | End index | Symbolic Value | Binary Encoding |
|---|---|---|---|---|
| *mode* | 0 | 2 | *manual* | *[f, f]* |
| *mode* | 0 | 2 | *cruise* | *[f, t]* |
| *mode* | 0 | 2 | *disengaged* | *[t, f]* |
| *speed* | 2 | 4 | *slow* | *[f, f]* |
| *speed* | 2 | 4 | *cruise* | *[f, t]* |
| *speed* | 2 | 4 | *fast* | *[t, f]* |
| *button* | 4 | 5 | *on* | *[t]* |
| *button* | 4 | 5 | *off* | *[f]* |

**Table 2: Output Data Types for the Cruise Controller**

essential for infinite data types. (Though see our remarks in Conclusions Section 6)

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

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

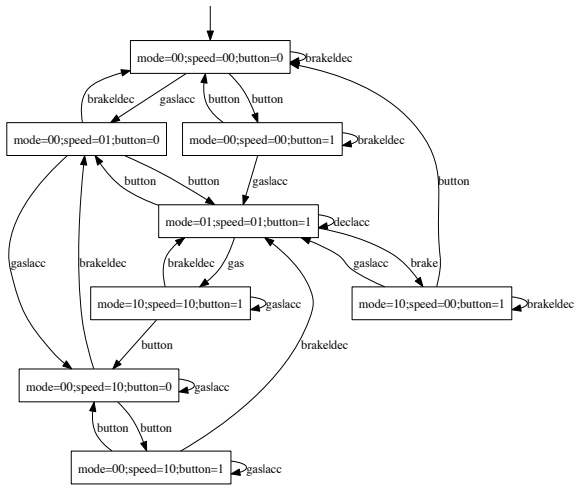


**Figure 1: Cruise Controller State Diagram**

# 4. CASE STUDIES IN LEARNING-BASED TESTING

In this section we present two industrial case studies which were tested with *LBTest*. These are: i) an access server from Fredhopper (FAS), and ii) a break-by-wire (BBW) system from Volvo Technology. Both these case studies represent mature applications from quite different industrial domains yet the tool was able to find errors in both of them, which is a promising achievement. The following features are important in these case studies:

- The FAS is an e-Commerce application while the BBW is an embedded application from the automobile industry.
- The FAS has been developed and evolved over 12 years and its various modules have been tested with automated and manual techniques. The BBW is relatively newer and is not widely adopted yet.
- The FAS does not have very strict timing constraints. The BBW has very strict timing constraints to ensure the safety of the vehicle.
- The FAS is very complex in terms of the input alphabet and lines of code. The BBW is smaller on both these counts.
- The FAS only involves events and finite data types while the BBW involves events and infinite data types.

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

## 4.1 Case Study 1: Access Server

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

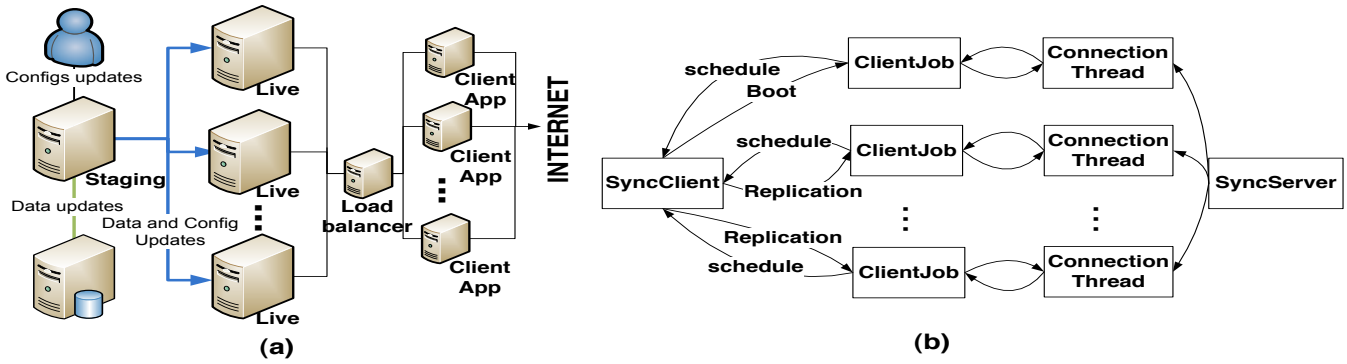


**Figure 2: (a) an FAS Deployment and (b) Interactions in the Replication System**

| Constant Symbol | Encodings |
|---|---|
| *setAcceptor* | *a* |
| *schedule* | *b* |
| *searchJob* | *c* |
| *businessJob* | *d* |
| *dataJob* | *e* |
| *connectThread* | *f* |
| *noConnectionThread* | *g* |

**Table 3: FAS Input Data Type**

as their contents, while SyncClient receives data and configuration updates according to the schedule. Fig. 1(b) shows the interactions in the Replication System. Informally, the Replication Protocol is as follows: the SyncServer begins by listening for connections from SyncClients. A SyncClient creates and schedules a ClientJob object with job type Boot that connects immediately to the SyncServer. The SyncServer then creates a ConnectionThread to communicate with the SyncClient's ClientJob. The ClientJob asks the ConnectionThread for replication schedules, notifies the SyncClient about the schedules, receives a sequence of file updates according to the schedule from the ConnectionThread and terminates.

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

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

$$\Sigma_{in} = \{setAcceptor,\ schedule,\ searchjob,\ businessjob,\ datajob,\ connectThread,\ noConnectionThread\}$$

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

**Requirement 1**: *If the SyncClient is at state* `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]

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

**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`.

$$G(jobtype = Boot \ \wedge \ state = WaitToBoot \wedge in = connectThread \rightarrow X(jobtype = Boot \wedge state = Booting))$$

**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`}*).*

$$G(jobtype = Boot \wedge state = Booting \wedge in = schedule \rightarrow X(schedules = \{data,\ business,\ search\} \wedge state = WaitToReplicate))$$

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

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

| Data Type | Start Index | End Index | Symbolic Output | Binary Encoding | Description |
|---|---|---|---|---|---|
| schedules | 0 | 3 | $\phi$ | *[f, f, f]* | Specifies the replication schedules to which the `SyncClient` should commit at any time. |
| schedules | 0 | 3 | `{search}` | *[f, f, t]* | |
| schedules | 0 | 3 | `{business}` | *[f, t, f]* | |
| schedules | 0 | 3 | `{business, search}` | *[f, t, t]* | |
| schedules | 0 | 3 | `{data}` | *[t, f, f]* | |
| schedules | 0 | 3 | `{data, search}` | *[t, f, t]* | |
| schedules | 0 | 3 | `{data, business}` | *[t, t, f]* | |
| schedules | 0 | 3 | `{data, business, search}` | *[t, t, t]* | |
| state | 3 | 6 | `Start` | *[f, f, f]* | Specifies the state which the `SyncClient` is in as specified by the `SyncClient` State Machine. |
| state | 3 | 6 | `WaitToBoot` | *[f, f, t]* | |
| state | 3 | 6 | `Boot` | *[f, t, f]* | |
| state | 3 | 6 | `WaitToReplicate` | *[f, t, t]* | |
| state | 3 | 6 | `WorkOnReplicate` | *[t, f, f]* | |
| state | 3 | 6 | `End` | *[t, f, t]* | |
| jobtype | 6 | 9 | `nojob` | *[f, f, f]* | Specifies the type of client job scheduled by the `SyncClient` according to the replication schedules received. |
| jobtype | 6 | 9 | `Boot` | *[f, f, t]* | |
| jobtype | 6 | 9 | `SR` | *[f, t, f]* | |
| jobtype | 6 | 9 | `BR` | *[f, t, t]* | |
| jobtype | 6 | 9 | `DR` | *[t, f, f]* | |
| files | 9 | 10 | `readonly` | *[f]* | Specifies whether the file system be written to by the `SyncClient`. |
| files | 9 | 10 | `writable` | *[t]* | |

**Table 4: `SyncClient` Output Data Types**

$$G(state = WaitToReplicate \ \wedge \ in = connectThread \rightarrow (jobtype = SR \rightarrow X(jobtype = SR \ \wedge \ state = WorkOnReplicate)) \wedge (jobtype = BR \rightarrow X(jobtype = BR \ \wedge \ state = WorkOnReplicate)) \wedge (jobtype = DR \rightarrow X(jobtype = DR \ \wedge \ state = WorkOnReplicate)))$$

**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.*

$$G(state \in \{WaitToReplicate, WaitToBoot\} \wedge in = noConnectionThread \rightarrow X(state = End))$$

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

$$G(state = End \rightarrow X(state = End))$$

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

$$G(state \neq End \rightarrow serach \in schedules \rightarrow (F(jobtype = SR) \ U \ state = End) \wedge business \in schedules \rightarrow (F(jobtype = BR) \ U \ state = End) \wedge data \in schedules \rightarrow (F(jobtype = DR) \ U \ state = End))$$

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

$$G(state = WorkOnReplicate \rightarrow X(files = writable \ U \ state \in \{End, \ WaitToReplicate\}) \wedge state \neq WorkOnReplicate \rightarrow X(files = readonly \ U \ state = WaitOnReplicate))$$

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

$$G(state = WorkOnReplicate \wedge in = schedule \rightarrow (search \notin schedules \wedge jobtype = SR \rightarrow X(search \in schedules)) \wedge (business \notin schedules \wedge jobtype = BR \rightarrow X(business \in schedules)) \wedge (data \notin schedules \wedge jobtype = DR \rightarrow X(data \in schedules)))$$

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

$$
\begin{array}{c}
G(search \in schedules \wedge \\
state \in \{WorkOnReplicate,\ Booting\} \wedge \\
F((state = WaitToReplicate \wedge \\
jobtype = nojob\ \wedge\ in = searchjob) \rightarrow \\
X(jobtype = SR\ \wedge\ search \notin schedules)) \wedge \\
F((state = WaitToReplicate \wedge \\
jobtype = nojob\ \wedge\ in = businessjob) \rightarrow \\
X(jobtype = BR\ \wedge\ business \notin schedules)) \wedge \\
F((state = WaitToReplicate \wedge \\
jobtype = nojob\ \wedge\ in = datajob) \rightarrow \\
X(jobtype = DR\ \wedge\ data \notin schedules)))
\end{array}
$$

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

### 4.1.1 Discussion of Errors Found

Nine out of eleven requirements were passed. For requirements 8 and 9, *LBTest* gave warnings (due to a loop in the counterexample) corresponding to tests of liveness requirements that were never passed. The counterexample for both these requirements was "setAcceptorSchedulebusinessJobbusinessJob". After the symbol *"businessJob"* a loop starts in the counterexample which has been unfolded just once. This violates requirement 8 because if we keep reading *businessJob* from here the SUT does not go to the end state as specified. This violates requirement 9 because we are in the start state after reading this sequence rather than *WaitOnReplicate* or *End* states as specified. We do not reach any of these if we keep reading *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 error in the user requirements. However, *LBTest* continued to produce warnings for requirement 8, corresponding to a true SUT error. So in this case study *LBTest* functioned to uncover errors both in the user requirements and in the SUT.

## 4.2 Case Study 2: Brake-by-Wire

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

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

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

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

The infinite real valued data types can be addressed by defining discretisation formulas as described in Section 3.2. Therefore, an SUT wrapper is required to discretise the real-valued inputs and outputs of BBW into alphabets of finite symbols. The real-valued inputs are discretised into a set of four input events given by

$$\Sigma_{in} = \{brake, acc, accbrake, none\}$$

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

The discretisation formula for each symbolic output is also shown in column four of Table 7. Note that in Table 7, $vehSpeed_i$ 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

| PLTL Requirement | Verdict | Total Testing Time (hours) | Hypothesis size (states) | Model checker queries | Learning queries | Random queries |
|---|---|---|---|---|---|---|
| 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 |

**Table 5: Performance of *LBTest* on Fredhopper Access Server case study**

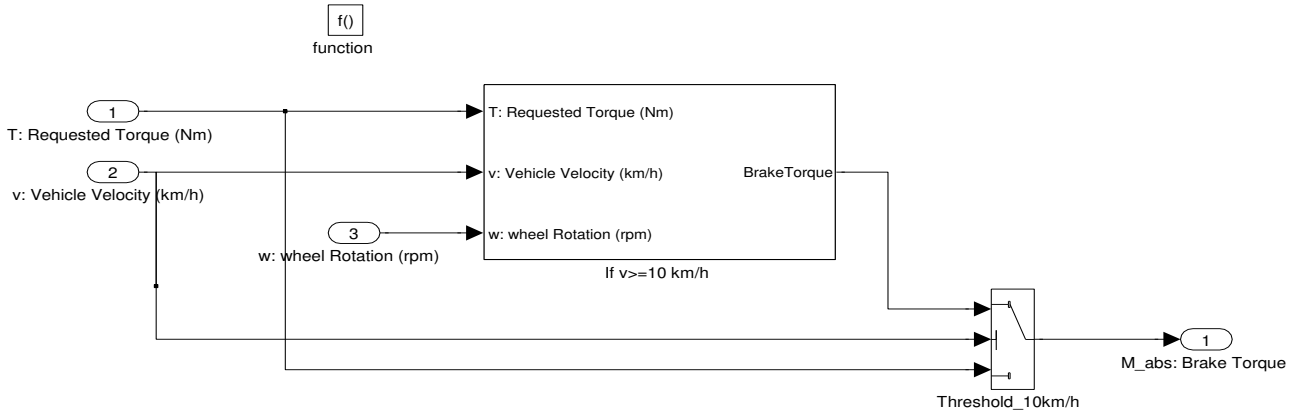


**Figure 3: A High Level Simulink Model of BBW System**

| Constant Symbols | Encodings |
|---|---|
| *brake* | *a* |
| *acc* | *b* |
| *accbrake* | *c* |
| *none* | *d* |

**Table 6: BBW Input Data Type**

$km/h$ 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 is essential to calculate the slip rate of the wheels. The slip rate denoted by *slip* in Table 7 of a wheel (e.g front right denoted by FR) is the ratio of the difference of $vehSpeed - wSpeedFR$ and the vehicle speed $vehSpeed$. The vehicle is considered slipping when the slip rate $slip > 0.2$ otherwise the vehicle is considered not slipping.

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

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

$$G(\, in = brake \rightarrow F(wheel = wheelRotateFR \rightarrow torque \; = \; torqueOnFR)\,)$$

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

$$G(\, (slip = slipFR \; \wedge \; speed = vehicleMove \wedge in = brake) \rightarrow torque \; \neq torqueOnFR\,)$$

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

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

### 4.2.1 Discussion of Errors Found

| Data Type | Start Index | End Index | Discretisation Formula | Symbolic Output | Binary Encoding |
|---|---|---|---|---|---|
| speed | 0 | 2 | $vehSpeed \leq 10$ | `vehicleStill` | *[f, f]* |
| speed | 0 | 2 | $vehSpeed > 10$ | `vehicleMove` | *[f, t]* |
| speed | 0 | 2 | $vehSpeed_i < vehSpeed_{i-1} \wedge vehSpeed_{i-1} > 0$ | `vehicleDecreased` | *[t, f]* |
| wheel | 2 | 4 | $\omega SpeedFR > 0$ | `wheelRotateFR` | *[f, f]* |
| wheel | 2 | 4 | $\omega SpeedFL > 0$ | `wheelRotateFL` | *[f, t]* |
| wheel | 2 | 4 | $\omega SpeedRR > 0$ | `wheelRotateRR` | *[t, f]* |
| wheel | 2 | 4 | $\omega SpeedRL > 0$ | `wheelRotateRL` | *[t, t]* |
| torque | 4 | 6 | $torqueOnFR > 0$ | `torqueFR` | *[f, f]* |
| torque | 4 | 6 | $torqueOnFL > 0$ | `torqueFL` | *[f, t]* |
| torque | 4 | 6 | $torqueOnRR > 0$ | `torqueRR` | *[t, f]* |
| torque | 4 | 6 | $torqueOnRL > 0$ | `torqueRL` | *[t, t]* |
| slip | 6 | 8 | $10 * (vehSpeed - \omega SpeedFR) > 2 * vehSpeed$ | `slipFR` | *[f, f]* |
| slip | 6 | 8 | $10 * (vehSpeed - \omega SpeedFL) > 2 * vehSpeed$ | `slipFL` | *[f, t]* |
| slip | 6 | 8 | $10 * (vehSpeed - \omega SpeedRR) > 2 * vehSpeed$ | `slipRR` | *[t, f]* |
| slip | 6 | 8 | $10 * (vehSpeed - \omega SpeedRL) > 2 * vehSpeed$ | `slipRL` | *[t, t]* |

**Table 7: Brake by Wire Output Types**

| PLTL Requirement | Verdict | Total Testing Time (sec) | Hypothesis Size (States) | Model Checker Queries | Learning Queries | Random Queries |
|---|---|---|---|---|---|---|
| Req 1 | Pass | 2065 | 11 | 0 | 1501038 | 150 |
| Req 2 | Fail | 58 | 537 | 18 | 34737 | 2 |
| Req 3 | Pass | 960 | 22 | 0 | 1006275 | 130 |

**Table 8: Performance of *LBTest* on Brake By Wire case study**

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

## 4.3 Conclusions from Case Studies

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

# 5. COVERAGE

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

## 5.1 Requirements Coverage Metrics for LBTest

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

| Coverage Criteria | % Coverage | No. of $TC_{max\ cov}$ | Total TCs |
|---|---|---|---|
| State | 100 | 149 | 2437 |
| | 100 | 17 | 2650 |
| | 100 | 27 | 2424 |
| Transition | 100 | 797 | 4570 |
| | 100 | 634 | 2740 |
| | 100 | 112 | 2866 |
| Edge Pair | 100 | 813 | 2919 |
| | 97.5 | 3370 | 3370 |
| | 100 | 2096 | 2400 |
| Prime Paths | 44.8 | 2559 | 2559 |
| | 44.8 | 2251 | 2251 |
| | 51.7 | 2997 | 2997 |

**Table 9: Coverage Metrics for LBTest**

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

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

Table 9 shows the coverage achieved by LBTest for each of the above mentioned criteria. The first column shows the coverage criteria used. Each experiment was repeated three times to take into account the varying influence of random test cases. The results for each experiment were recorded in three different rows against each coverage criteria. The second column shows the percentage of coverage achieved. The third column shows the minimum number of *test cases(TCs)* executed to reach the maximum coverage for each criteria and the last column shows the total number of test cases (TCs) executed for completely learning the SUT.

### 5.1.1 State Coverage

Table 9 shows the state coverage achieved by LBTest in three different experiments. In the actual CC state diagram there are 8 states but the learning algorithm learns a minimal model of the system which is behaviourally equivalent to the actual system. We get 100% state coverage in this case for three different experiments because the random, model checker and active learning queries visit all the states of the SUT.

### 5.1.2 Transition Coverage

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

### 5.1.3 Edge-Pair Coverage

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

### 5.1.4 Prime Path Coverage

Another coverage criteria given in [4] is *prime path coverage.* This is a rather strict coverage criteria as it does not allow any internal loops inside the path. Therefore the coverage for this particular metric is the lowest as compared to other metrics given in Table 9. We first wrote code to find all prime paths in CC state diagram and the number of these paths was found to be 87. But LBTest at best was able to cover roughly 50% of these prime paths. The reason is the same as observed in the previous section that to learn a behaviourally equivalent representation of the SUT it is not necessary to cover all prime paths in the SUT. In this case we observe that 100% prime path coverage has significant redundancy when compared with LBT.

## 6. CONCLUSIONS AND FUTURE WORK

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

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

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

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

## 7. REFERENCES

[1] The conformiq designer tool

(http://www.conformiq.com/products/conformiq-designer/).

[2] Leirios test designer tools (http://www.leirios.com).

[3] Webdriver (http://code.google.com/p/selenium/).

[4] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[5] P. Chauhan, E. M. Clarke, J. H. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In *Proc. 14th International Conference On Formal Methods in Computer-Aided Design (FMCAD02)*, 2002.

[6] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model verifier. In *Proceeding of International Conference on Computer-Aided Verification (CAV'99)*, number 1633 in LNCS. Springer, 1999.

[7] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.

[8] M. Broy et al. (eds). *Model-based Testing of Reactive Systems, LNCS 3471*. Springer, 2005.

[9] G. Fraser, F. Wotawa, and P.E. Ammann. Testing with model checkers: A survey. Tech. rep. 2007-p2-04, TU Graz, 2007.

[10] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. *Logic Journal of the IGPL*, 14(5):729–744, 2006.

[11] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of abstract data types*. Wiley, 1996.

[12] K. Meinke. Automated black-box testing of functional correctness using function approximation. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 143–153, New York, NY, USA, 2004. ACM.

[13] K. Meinke and F. Niu. A learning-based approach to unit testing of numerical software. In *Proc. Twenty Second IFIP Int. Conf. on Testing Software and Systems (ICTSS 2010)*, number 6435 in LNCS, pages 221–235. Springer, 2010.

[14] K. Meinke and F. Niu. Learning-based testing for reactive systems using term rewriting technology. In *Proceedings of the 22nd IFIP International Conference on Testing Software and Systems*, volume 7019 of *LNCS*, pages 97–114, Berlin, Heidelberg, 2011. Springer.

[15] K. Meinke, F. Niu, and Muddassar A. Sindhu. Learning-based software testing: a tutorial. In *Proc. Fourth Int. ISoLA workshop on Machine Learning for Software Construction*, number 336 in CCIS, pages 200–219. Springer, 2012.

[16] K. Meinke and Muddassar A. Sindhu. Lbtest: A learning-based testing tool for reactive systems. To appear in proc ICST, IEEE Computer Society, 2013.

[17] Karl Meinke and Muddassar A. Sindhu. Incremental learning-based testing for reactive systems. In Martin Gogolla and Burkhart Wolff, editors, *Tests and Proofs*, volume 6706 of *Lecture Notes in Computer Science*, pages 134–151. Springer, 2011.

[18] D. Peled, M.Y. Vardi, and M. Yannakakis. Black-box checking. In *Formal Methods for Protocol Engineering and Distributed Systems FORTE/PSTV*, pages 225–240. Kluwer, 1999.

[19] H. Raffelt, B. Steffen, and T. Margaria. Dynamic testing via automata learning. In *Hardware and Software: Verification and Testing*, number 4899 in LNCS, pages 136–152. Springer, 2008.

[20] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc, San Francisco, CA, USA, 2006.

[21] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In *Formal Methods and Testing*, pages 39–76, 2008.

[22] E. Weyuker. Assessing test data adequacy through program inference. *ACM Trans. Program. Lang. Syst*, 5(4):641–655, 1983.

[23] M.W Whalen, A. Rajan, and M.P.E. Heimdahl. Coverage metrics for requirements-based testing. In *Proceedings of International Symposium on Software Testing and Analysis*, pages 25–36. ACM, July 2006.