Testing Safety Critical Avionics Software Using LBTest

Sebastian Stenlund

Supervisors : Ahmed Rezine
Karl Meinke
Examiner : Ahmed Rezine
Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/her own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Sebastian Stenlund
Abstract

A case study for the tool LBTest illustrating benefits and limitations of the tool along the terms of usability, results and costs. The study shows the use of learning based testing on a safety critical application in the avionics industry. While requiring the user to have theoretical knowledge of the tools inner workings, the process of using the tool has benefits in terms of requirement analysis and the possibility of finding design and implementation errors in both the early and late stages of development.
Acknowledgments

Thanks to Karl Meinke Professor at KTH, Ahmed Rezine Assistant Professor at LiU, Håkan Anderwall and Christoffer Nylén at Saab AB.
# Contents

1 Introduction ................................................. 1  
1.1 Motivation and background .............................. 1  
1.2 Formal methods in avionics ............................ 1  
1.3 Personal motivation and target group .................. 2  
1.4 Question of the thesis ................................. 2  
1.5 Method .................................................. 2  
1.6 Limitation and delimitation ............................ 3  
1.7 Structure of the report ................................. 3  

2 Theory ..................................................... 4  
2.1 Software development and software quality .......... 5  
2.2 Software Testing and development life cycle ........ 5  
2.3 Different testing methods ............................. 6  
2.4 Testability and what we can observe ................. 6  
2.5 Coverage .............................................. 7  
2.6 Structural- and requirements based testing ........ 7  
2.7 Automation of test case generation ................. 8  
2.8 Requirements ......................................... 8  
2.9 Formal methods and Formal specification ........... 8  
2.10 Finite State machine ................................ 9  
2.11 Learning a Finite State machine .................... 9  
2.12 Model checking ...................................... 9  

3 Learning based testing ..................................... 10  
3.1 Propositional Linear Temporal Logic ................. 10  
3.2 Problems with model checking ......................... 12  
3.3 Abstraction and infinite-state systems ............. 12  
3.4 SMV and NuSMV ................................... 12  
3.5 Learning-Based Testing ............................... 13  

4 The LBTest Tool ........................................... 14  
4.1 Interface and configuration files .................... 14  
4.2 L* and IKL learning algorithms ...................... 14  
4.3 BDD and BMC model checkers ......................... 15  
4.4 LBTest SUT communication ............................ 15  

ii
5 Case study
  5.1 Requirements analysis ........................................... 17
  5.2 Requirement analysis ........................................... 18
  5.3 The SUT input and sensory data ..................................... 19
  5.4 LBTest integration .................................................. 19
  5.5 The Wrapper ......................................................... 19
  5.6 Sampling time or sequential abstraction .......................... 20
  5.7 Different wrappers .................................................. 21
  5.8 Learning algorithms and hard to learn models ................. 22

6 Results ................................................................. 24
  6.1 LBTest results ...................................................... 24
  6.2 Testing environment ................................................ 25
  6.3 Low level testing ................................................... 25
  6.4 Writing requirements .............................................. 25
  6.5 Writing wrappers .................................................. 26

7 Discussion ............................................................. 27
  7.1 Integrating LBTest .................................................. 27
  7.2 LTL and Model Checking ........................................... 27
  7.3 Learning algorithms ............................................... 28
  7.4 Wrappers ............................................................... 28
  7.5 Previous case studies ............................................... 29
  7.6 Summary .............................................................. 29

Appendices ............................................................... 30
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Software development life-cycle.</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Modules and sub modules abstract view.</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>Ec (equivalence classes) and edge cases (stars at 9, 10 and 11) around Threshold T</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>A simple state machine, inputs(numbers) outputs (characters). Counting from A to C when given sequences of 1s and resetting to A given a 0.</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>LBTTest simplified view.</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>A visual representation of a systems initial state and subsequent branches(paths) of states and their output.</td>
<td>11</td>
</tr>
<tr>
<td>3.3</td>
<td>The abc counter from the theory chapter unrolled.</td>
<td>12</td>
</tr>
<tr>
<td>4.1</td>
<td>LBTTest functionality, Queries becomes inputs to the SUT.</td>
<td>15</td>
</tr>
<tr>
<td>5.1</td>
<td>Possible state transitions used to derive the requirements (not a correct state chart).</td>
<td>18</td>
</tr>
<tr>
<td>5.2</td>
<td>Abstract view of the input and output of the sub module being tested.</td>
<td>19</td>
</tr>
<tr>
<td>5.3</td>
<td>Simplified wrapper pseudo-code.</td>
<td>20</td>
</tr>
<tr>
<td>5.4</td>
<td>Possible behaviours in between observation points O1 and O2.</td>
<td>21</td>
</tr>
<tr>
<td>5.5</td>
<td>Example of a counter and a possible &quot;hard to learn&quot; state X</td>
<td>23</td>
</tr>
<tr>
<td>6.1</td>
<td>LBTTest with 6 of the 24 requirements</td>
<td>24</td>
</tr>
<tr>
<td>1</td>
<td>A: Example of wrapper output refinement(added speed as output) B: Example of multiple inputs(combinatorial)</td>
<td>31</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Logic operators used in LTL with propositions P and Q ........................................ 11
3.2 Future time LTL operators with propositions P and Q ........................................ 11

5.1 Events that should trigger state changes between aircraft modes. Short means less than one second, long means more than one second. immediate means change should happen at the next execution cycle ......................................................... 17
5.2 Possible state transitions from the state LANDING (a subset of all requirements) . 18
5.3 Events that should trigger state changes the aircraft mode ................................. 18
Chapter 1

Introduction

This chapter explains the motivation, background and purpose of this study.

1.1 Motivation and background

Formal methods and Model checking are popular topics in computer science research and have the potential to increase the effectiveness and correctness of computer software testing and modelling. SAAB Aeronautics[11] together with KTH provided the objective of this thesis in evaluating the test automation tool LBTest.

SAAB is a company that has about 15 000 employees and is active in both civil- and military areas world wide. Saab Aeronautics is a part of Saab that works with advanced development of military and civil aviation technology [11].

LBTest is being developed at KTH since 2011 as a platform for combining formal specifications, learning algorithms and model checking as a verification and test case generation tool[5] in a black box testing environment.

1.2 Formal methods in avionics

The avionics industry has high requirements for software quality in terms of reliability and safety. In order to meet these standards, the Radio Technical Commission for Aeronautics (RTCA) and the European Organization for Civil Aviation Equipment(EUROCAE) jointly developed the 1992 DO-178B standard (Software considerations in airborne systems and equipment certification)[10].

The DO-178B standard is used, among others, by the Federal Aviation Administration (FAA) and the European Aviation Safety Agency (EASA) as a guideline for certifying software based aerospace systems. 178C is an updated version of 178B and has included among other things DO-331: Model based development and verification, and DO-333: Formal methods as a complement to other software testing methods[10]. The 2011 DO-178C is a clarification and an update to DO-178B and is designed to meet up with new development methods and tools. The DO-178C standard also provides guidelines regarding model based development, object oriented & Related Technologies, and processes on verification[10][15].

Both DO-178B and DO-178C includes: test cases, code coverage, and requirements based testing and testing of executable object code. The addition of formal methods in 178Cs DO-333 is to provide further confidence in the correctness of the software, using mathematical models such as formal methods, and model checking[10]. A formal method approach that is emerging is combining learning based testing with model checking( more on this in chapter 2).
1.3 Personal motivation and target group

As a student and software developer testing has always fascinated me. Verifying that a software works as intended may arguably be the most important part of developing a system. Testing also affects all stages of the development lifetime. Proving that a program does work correctly is something that can, for non-trivial systems, never be done entirely. Finding methods that are effective and practical that get us closer to ensuring software correctness will always be an interesting subject.

This report is aimed to work as a practical example for industrial practitioners interested in Learning-Based Testing or students interested in modern testing methodologies, and to provide new research results relevant to the future of Learning-Based Testing and the tool LBTest.

1.4 Question of the thesis

The main question this thesis aims to explore is.

- Can learning based testing be used in an industrial environment?

1.5 Method

The study was done by studying literature for testing methods, formal methods and learning based testing. Thereafter provide a case study on a safety critical avionic software at Saab Aeronautics using the tool LBTest.

Steps in making the LBTest case study.

- Producing a requirement analysis of the documents and requirements related to the software being tested.
- Translating the requirements to formal specifications as input to LBTest.
- Configuring LBTest to work with the software.
- Analysing the test results.
- Evaluation and discussion of the process and results of the steps above.
1.6 Limitation and delimitation

This report will not provide a statistical analysis in running the testing tool LBTest on a large number of different programs or systems, but will instead provide a case study for implementing LBTest into an industrial application in the avionics industry.

While the software under test can be considered large, only a subset of its functionality was tested, also there are several approaches and methods when using LBTest and the tool itself is very customizable and flexible. Therefore one case study cannot cover all the possible tool configurations.

This study will focus mainly on high level requirement testing. The theory of model checkers and learning algorithms will only be presented from the view of learning based testing.

The LBTest tool did not find any errors in the software under test, and as of writing, neither did the traditional high level testing methods. This may be due to the maturity of the software and its development process or due to the model size of the abstracted model. Other case studies have shown that LBTest finds errors when the errors are injected into the system under test[2] and studies have also found errors in production software that has not been found before see previous case studies in the discussion chapter.

LBTest is not yet publicly available and some of the information in this thesis regarding usage of the tool and its configuration are referring to recommendations in the LBTest manual. The tool is also currently being developed, new features and functionality as well as changes to the graphical interface may be implemented.

Technical terms, words and values have been changed in order to conform to Saab’s information policy.

1.7 Structure of the report

Chapter 2 starts with software development theory regarding testing and learning based testing. Chapter 3 describes how Learning based testing in LBTest works and the semantics of formal specifications using linear temporal logic (LTL). Chapter 4 describes the configuration options and the use of the tool LBTest. Chapter 5 provides the case study for this thesis and Chapter 6 the results. Finally Chapter 7 finishes with a discussion on the findings of this study.
Chapter 2

Theory

This chapter begins with a brief presentation and explanation of key concepts and definitions used in software-engineering activities such as development and testing, and it is partially needed for the understanding of this study. More in-depth theory is provided on the subject of Learning-Based Testing and formal methods later in this chapter.

Software engineers need to solve the problem of producing high-quality software within reasonable time limits and costs. It is important to verify a system’s correctness in accordance with its requirements early in the development process, continuously throughout it, and during its maintenance [7][8]. Many methods for the development process have been invented and evaluated during the years including the Waterfall, Spiral, and agile models [8]. Choosing the right method is dependent on factors such as resources, team size, and the type of system being developed. What all development processes have in common however is the systems development life cycle. These stages include: planning, design, implementation, and maintenance see Figure 2.1.

The quality of a system may be measured differently and software quality characteristics have been defined in the ISO 9126[8] standard for information technology as: Functionality, Reliability, Usability, Efficiency, Maintainability and Portability. The importance of each characteristic varies with the purpose of the system and its requirements.
2.1 Software development and software quality

![Software development life-cycle](image)

Figure 2.1: Software development life-cycle.

2.2 Software Testing and development life cycle

Each stage of development is dependent on the earlier stages. Also, finding faults at a later stage is much more expensive than finding them at a earlier stage. Therefore testing is an important focus during all parts of the life cycle [7] [8]. When adding features to a program at the maintenance stage, development on that feature often has to start over from the planning stage with a new design of the feature and of the implementation. This also creates a need to again verify the unmodified parts and requirements of the systems to make sure they are not affected by the added feature (regression testing).
2.3 Different testing methods

Regression testing was mentioned before as the process of making sure that the legacy functionality of the software is still working when either corrections or new features are added to the system. Unit testing is the process of making sure a single “part” or sub module of a system works as intended. It relies on knowledge of the internals of the module and is usually done by the developer or developers [7] [8]. Integration testing is the process of testing sub modules connected together, or even a whole system. Integration testing can be carried out by an independent test group, the reason being that the test group will have their own idea of what to test and not be affected by the developers implementation or the internals of the system.

Other stages of testing include Usability testing, which determines how easy the system is to understand and use, Performance testing, that ensures that the performance of the system is within performance limits, and Acceptance testing where the end user uses the system and verifies that it has the requested functionality. When providing test cases it might be important to remember to test for erroneous input that is not expected in order to make sure that the software handles the error correctly. Testing for input outside of the normal input range like this is called robustness testing.

![Figure 2.2: Modules and sub modules abstract view.](image)

In testing the difference between modules and sub modules is not clear cut, it depends largely on the system design and implementation. A larger sub module may have sub modules of its own see fig 2.2.

2.4 Testability and what we can observe

Whether a system can be tested or not has to do with factors such as observability [6], is it possible to read the internal state of the system? If not how can we test it? A lot of times additional code has to be written just to make testing possible. This is further complicated if there are a lot of connections between modules working together (high coupling). With high coupling the problem that needs to be solved is the dependency between the modules. The functionality of one or more sub modules may be dependent on other sub modules that have not yet been implemented or tested. Writing “stubs” in place of functionality of these modules is a common way of getting rid of dependencies when testing. Writing manual stubs takes time but this time can be reduced if the order of developing and testing modules is taken into consideration. With the introduction of Object Oriented Programming OOP, there is an increasing need for more research in this area [7].
2.5 Coverage

Test cases are written to test specific behaviours of a system. It is impractical to cover all possible input combinations a system can receive. A way to counter this is to identify what ranges or equivalence classes [7] of input data makes (or should make) the system behave differently and select those data sets. For example, a function may change its behaviour when the value it receives is above a certain threshold (T). If the requirement of the function is to turn on the light when a value goes above 10, the tester could choose any number that is above 10 to effectively represent all those test cases. Equivalence class test cases could then choose to test arbitrary numbers like the values 5 and 15 for representing above and below the threshold.

To make sure that the transition is exactly on 10 the tester could make test cases that also test values as close to the threshold as possible (9, 10 and 11), this is called edge case coverage or boundary value analysis (see fig. 2.3 below). Edge case coverage is important since these kinds of edges are a large source of both implementation and specification errors[7].

Test cases should also be written to cover the specified requirements and should provide confidence not only that bad behaviour will not happen (i.e. safety), but also that the specified correct behaviour does happen (i.e. liveness).

![Figure 2.3: Ec (equivalence classes) and edge cases (stars at 9, 10 and 11) around Threshold T](image)

Coverage in this case is mostly related to structural testing and not on black box requirements based testing. Requirements based coverage measures are not well defined [2].

2.6 Structural- and requirements based testing

Structural- or White Box Testing is the process of testing the internal details of a system and requires knowledge and access to those internal details [7] [8]. White box testing is useful for coverage of data- and control flow, branch testing, statement coverage and path testing. An added benefit of this knowledge of internal data types, algorithms and structures, may provide information on their respective limits. For example the point of overflow of data types like integers and floating point numbers used, may indicate a higher or lower boundary that needs to be taken into consideration and could easily be missed otherwise. How much is left to the implementer or by the designer may differ between projects.

One disadvantage of white box testing is that while coverage of the implementation is thorough, the coverage of the actual specified requirements might not be fully evaluated [7]. This is further complicated if the designer of the requirements and the implementer have different ideas of what the requirements specify. When testing against the requirements one often uses Black box testing where there is no (or should not be any) knowledge of the system internals. Black box testing needs the requirements to be well specified and cover wanted and unwanted behaviour of the system.

Grey box testing combines the requirement based testing of black box testing with partial knowledge of the internals of the system. The aim of Grey box testing is to cover the advantages of both of the above testing methods, while eliminating their disadvantages [8].

Conformance testing is the process of determining if a given system conforms to a reference model. The term is used on both white and black box testing [3].

Combinatorial testing is a black-box method concerned with input data generation for several inputs at the same time, and their combinations. Testing all possible combinations exhaustively
is impractical and often leads to combinatorial explosions[13]. Combinatorial testing has different methods of generating sets of input like n-wise or pairwise testing.

2.7 Automation of test case generation

The process of testing has to take a reasonable amount of time and effort. To help with this Automation is of importance. Automation makes the process of testing less prone to human error than manual testing [7] and may also be very cost effective. One disadvantage is that human knowledge and insight may be lost [8]. Automatic test case generation (ATCG) may also help with reducing test redundancy and increasing test coverage.

2.8 Requirements

In development and testing, designers, developers, and testers need to have the same idea on what the system's functionality should be. Therefore well specified, detailed and understandable requirements should be specified and maintained [7] [8]. A balance must be made between textual in formal specifications and formal specifications. An alternative is to have documentation of both. In the latter case, care must be taken to make sure that the informal and the formal specification conform to one another. Another advantage of having formal specifications is that no ambiguity of natural language exists and that automation of requirements based test case generation may be possible.

High level requirements(HLR) specify what the software or system should do, low level requirements(LLR) specify how the high level requirements should be met on implementation. High level requirements are created at an early stage of development while low level requirements are available at later stages.

Since LLR are based on HLR, it is important that the HLR requirements are correct. Both HLR- and LLR testing are needed and finding errors in the HLR early in development may save development costs.

2.9 Formal methods and Formal specification

Verifying that a system works correctly is often hard to prove, how do we know that the test cases cover everything that needs to be covered? Or even all of the requirements completely? Proving software correctness is important in all systems but perhaps even more so in safety critical systems where mistakes may endanger human lives [7] [8]. Protocols, standards and guidelines have been developed to make the process of testing more thorough. Formal methods may be used to provide mathematical proof of correctness in certain areas [7] [4], and in others provide more confidence in the correctness of the system.

A formal specification can provide an abstract mathematical model of a systems behaviour. Using a formal specification the designer specifies the systems requirements with a formal language; formal verification can then be automated to prove correctness according to the systems specification. One way of creating such a mathematical model is using a finite state machine [5]. Other formal specifications include programming languages, logic formulas and other kinds of unambiguous languages that can be used to describe behaviours and relations[3].
2.10 Finite State machine

Finite state machines (FSM) or finite state automata are mathematical models of a set of states and possible transitions between them[14] given a set of inputs. This method of representation may be be used to (among other things) model sequential logic and behaviour such as computer software. In finite state machines a given state may transition to another state given a certain input and provide some output[1] [5] see fig 2.4. Finite state machines are used by model checkers for representations of software models. Comparing such models with model-checkers against functional requirements for verifying system behaviour is an effective process for proving software correctness. Building and maintaining such a model normally includes much manual work, especially for more complex systems. Also this process may be prone to human errors and its possible that the model does not correctly represent the actual system behaviour.[4]

![Figure 2.4: A simple state machine, inputs(numbers) outputs (characters). Counting from A to C when given sequences of 1s and resetting to A given a 0.](image)

2.11 Learning a Finite State machine

Algorithms to learn finite state machines automatically have been developed and one such algorithm is the Angluins L* algorithm[12]. A learning algorithm will send sequences of signals called traces or “strings of input” to a so called teacher in this case the System Under Test (SUT) and read the response output string to learn the behaviour of the machine. When finished it will ask an oracle if the learnt model was correct. If the model was incorrect the oracle sends back an input string that shows where the model does not conform to the real system with a counter-example trace[5].

2.12 Model checking

A model checker (MC)[1] will take as input an automaton based model of a system(SUT)), as well as a temporal logic property (requirement). If the model checker finds any violation to the property in the model it will produce a counter example or witness. The counter example is a trace of inputs to the system that will violate the property and is therefore a quick example to the developer on where and when the problem appears. If no counter-example is produced there is proof that the property is not violated in the given model. Model checkers have also been suggested for automatic test case generation( ATCG ) [4] [5]. There are different approaches to model checking that have been developed during the years, each with its own advantages and disadvantages. One feature of the tool LBTTest is the possibility of switching between available model checkers [5].
Chapter 3

Learning based testing

Learning based testing combines learning algorithms with model checkers and by doing so automates the task of creating and managing a model, the model does not need to be manually updated and is learnt from the system’s observed behaviour. While learning the model LBTest continuously uses the model checker(oracle) to check for coherence between the model and the specified requirements. This has an advantage in that the whole model does not need to be learnt to find an incoherence to the specification or errors in the software. The counter-examples made from the model checker will help the learning algorithm better learn the system[5] see fig 3.1.

![Diagram of LBTest](image)

Figure 3.1: LBTest simplified view.

3.1 Propositional Linear Temporal Logic

Propositional Linear temporal logic (PLTL a.k.a. LTL or PTL [3]) is used to express requirements and behaviour of a system. LTL uses propositional logic operators with the addition of temporal operators. Temporal operators state when in time (or sequence) the logic expression has to be
true. Similarly to propositional logic, LTL formulas can often be written in many different ways using different operators [3].

<table>
<thead>
<tr>
<th>Logic</th>
<th>Logic operator</th>
<th>LTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implies</td>
<td>→</td>
<td>P → Q if P then Q</td>
</tr>
<tr>
<td>Not</td>
<td>¬</td>
<td>¬P negation</td>
</tr>
<tr>
<td>And</td>
<td>∧</td>
<td>P &amp; Q and</td>
</tr>
<tr>
<td>Or</td>
<td>∨</td>
<td>P ∨ Q or</td>
</tr>
<tr>
<td>Xor</td>
<td>⊕</td>
<td>P XOR Q exclusive or</td>
</tr>
<tr>
<td>Equivalence</td>
<td>↔</td>
<td>P ↔ Q logical equivalence</td>
</tr>
</tbody>
</table>

Table 3.1: Logic operators used in LTL with propositions P and Q

<table>
<thead>
<tr>
<th>Future Time PLTL</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(P)</td>
<td>Next</td>
</tr>
<tr>
<td>G(P)</td>
<td>Globally</td>
</tr>
<tr>
<td>F(P)</td>
<td>Future</td>
</tr>
<tr>
<td>P U Q</td>
<td>Until</td>
</tr>
<tr>
<td>V(P) or W(p)</td>
<td>Releases or weakly until</td>
</tr>
</tbody>
</table>

Table 3.2: Future time LTL operators with propositions P and Q

The LTL operators are the logical: AND, OR, NOT, if → and iff ↔(iff) (if and only if) with the addition of temporal operators. LTL can express a future sequence of states on an infinite path. Operators include the global (G), next(X), future (F), release (V), Until (U) see table 3.1 and 3.2. Using the above operators, requirements for safety and liveness can be expressed.

For example the safety requirement that the output of a system should never output the symbol “error” can be expressed as: G(output != H). Another example is G(output = B → X(output = C)), note that some state transitions are conditional and some are not and will happen regardless of input see figure 3.2.

Figure 3.2: A visual representation of a systems initial state and subsequent branches(paths) of states and their output.

Past time operators are also available in NuSMV[9]. The requirements can often be expressed with the future time LTL operators. Past time operators can however provide flexibility in ways of expressing the formulas. Another rule is that when not specifying any temporal operators, time 0 is implicitly stated, this can be used to describe the initial state.

Expressing the liveness requirement that when pushing the exit button the program should always terminate may be expressed in LTL as G(button = pushed → F(program = terminate)). Here future means that it does not have to be the immediate next state, it only needs to happen eventually.

1 The Difference between U and V is that with U the state Q has to happen. In V the original statement has to be true forever if Q never happens.
sometime in the future, and this checks that the system will always follow the button press with an 
exit. No loop or deadlock should hinder the program from termination. If the requirement instead 
requires that the program should exit immediately after the button has been pressed the next(X) 
operator may be used instead of $F [\delta]$. A requirement that says that “$a$” should be true and stay 
true until “$b$” occurs may be defined as: $(a \cup b)$. If there is no requirement that $b$ must happen 
the $V$ operator may be used instead of $U$. see table 3.2

To illustrate the difference between an unrolled sequential state diagram and a finite state 
machine diagram see the difference between figure 2.4 and figure 3.3 below. The sequential state 
diagram shows the state $a$ as the initial state but also a recurring state whenever the program 
receives a 0 as input. the state $b$ is also shown as never being connected to another $b$ state in any 
path. Also illustrated is the fact that there is an infinite amount of possible paths when time or 
sequence is a factor. The system only truly resets when the system is restarted.

![Figure 3.3: The abc counter from the theory chapter unrolled.](image)

3.2 Problems with model checking

As the number of state variables of a system increases, the state space size grows exponentially. 
State space size of the system being analysed is a limiting factor on model checkers [1] [5]. Sev-
eral methods have been developed to counter this including abstraction, Binary decision diagrams 
(BDDs), bounded model checking algorithms and partial order reduction). Because of these ad-
vances in technology the size of models that can be used with model checking has increased to very 
large models able to accommodate many industrial applications.

3.3 Abstraction and infinite-state systems

A system may be of infinite-state size, which is often the case with data processing systems [1]. 
An abstraction may convert an infinite state machine into a finite one, or a finite state machine 
into a more manageable size. Input and output partitioning similar to equivalence classes may be 
used to reduce ranges of values into symbolic signals using sampling Time may also be abstracted 
depending on what we want to observe. For example observation of data originally from 100Hz 
down to 10Hz or 1Hz may significantly reduce the state space, and still accurately model the 
behaviour. Care needs to be taken that nothing we care about happens between those cycles due 
to under sampling.

3.4 SMV and NuSMV

NuSMV is a symbolic model checker developed by the Bruno Kessler foundation [9] and the 
Carnegie Mellon University. NuSMV is an implementation and extension to the SMV model
checker developed by McMillian at Carnegie Mellon University. NuSMV is able to check finite state system models against specifications and supports both LTL and CTL specification languages. Model checkers provided are both a Boolean satisfiability problem (SAT)-based Bounded Model Checker (BMC) and Binary Decision Diagram (BDD) based model checker. NuSMV is aimed at being an industrial standard, flexible and open platform for model checking.

3.5 Learning-Based Testing

Learning-Based testing is a heuristic approach for automated generation of test cases from formal specifications [3] [4]. Learning-Based testing uses a black box approach where it queries the SUT using a learning algorithm. The learning algorithm provides a model of the learned system and the learned model is then checked against the formal specification using a model checker. If the model checker finds an incoherence in the specification and the model it produces a counterexample. The counterexample is then used as an input to the SUT. If the counterexample violates the specified requirement property the system has been found faulty. If the counterexample does not violate the requirement the learning algorithm continues to refine the model. If the learning algorithm cannot find queries to learn more about the SUT it will ask a random input generator for a input sequence to find new reachable states in the SUT to work from.

Advantages of learning-based testing are that no model of the system is needed before-hand; instead the model is built on the fly. Another advantage of this is that the model created from the learning algorithm is known to represent the implemented behaviour of the system and not its specification. The learned model is instead checked against the specified requirements. If an error is found it is either the formal specification requirement or the implementation that is wrong.
Chapter 4
The LBTest Tool

LBTest is constantly being developed and new functionality and changes to the interface are made. While this thesis was written several new features were added, some on request from this project. Since the interface may change in LBTest, our focus is on how LBTest can be configured.

4.1 Interface and configuration files

The interface of LBTest shows the requirements being tested and their verdict (more on this in chapter 6). Output from NuSMV when the model checker is being run is displayed and several log files are available, among others are the graphical representation of the learnt automata. The visual representation as dot graphs (see appendix A and B) are a nice addition and are useful for debugging in the early stages of testing with LBTest. LBTest main purpose is not to create graphical representations and it is able to handle much larger models than would be possible to review using the graph or even render. Furthermore the model created by LBTest may not look as expected but can still provide the correct behaviour. How the model looks depends on the abstraction of the wrapper and the selected in- and outputs.

A configuration of how LBTest talks to the SUT is made using a text config file. The config file should contain where the SUT adapter / wrapper exists. The path to the model checker (NuSMV) the inputs available to the wrapper/SUT, and the possible outputs. Also the learning algorithm and model checker to be used are configured here. LBTest will also output the SMV (NuSMV model representation) file representing the last learnt model.

4.2 L⋆ and IKL learning algorithms

A learning algorithm is used to explore and learn possible paths and behaviour. The $L^*$ algorithm\cite{12} sends many queries to the teacher until a hypothesis is complete and then checks with the oracle (Model Checker) for coherence. Since counterexamples are produced only when the learning algorithm has a hypothesis ready, large systems may take a long time to produce an error. A solution to this is using the Incremental Kripke Learner algorithm (IKL) available in LBTest. IKL will more frequently ask the model checker for counter examples\cite{5} than $L^*$.

The advantage of querying the model checker more often is that on large models it may find errors more quickly. The time between two hypotheses models in $L^*$ can take a very long time. However $L^*$ may in the same time frame produce a more complete model that is closer to the behaviour of the system. In summary IKL may find errors faster than $L^*$, but $L^*$ may find a more accurate model in the same amount of time\cite{2}. 

14
4.3 BDD and BMC model checkers

The available BDD based method is significantly faster than the BMC approach but can only provide pass/warning verdicts[2]. BMC will find definite fail verdicts. A suggestion from developers of LBTest is to use the BDD checker first and then follow up warnings with the BMC checker[18][5].

After the learning algorithm and model checker are finished. The random input generator sends random input strings to explore the possibility of new states in the system that the learning algorithm did not find. If a new state is found the learning algorithm will build a new hypothesis again. String lengths and number of random queries are parameters that can be changed using the config file modifying the random sequence length and number of tries the random input generator runs. This determines when the LBTest tool finishes running and a larger setting may give more confidence in proving correctness of the software.

If no new trace in the model can be found for a certain number of random queries the test finishes saying that the model has been learnt completely. The random string length determines the length of each random trace.

4.4 LBTest SUT communication

LBTest talks to the wrapper by sending strings of comma separated symbols, input and output symbol names are configured in the config file, the actual value representation are implemented in the wrapper.

The wrapper responds to the input by sending back the output generated by each input see fig 4.1 The protocol works as follows, note that it may differ on the process results part depending on the learning algorithm used:

1. Start Wrapper executable (always starts on SUTs initial state)
2. Send strings of symbols (ex: “1,0,1” or “apple, pear”)
3. Read output string
4. Terminate the wrapper

**Process results part:**

5. If a new state was found or the learning algorithm can come up with a new string to test go to step 1.
6. Using the Model checker, compare the specified requirements model with the behavioural model and look for counterexamples in the specified requirements. If there is a counter-example use it as input in 1. if it has already been used continue.
7. If a counterexample has been used as input to the SUT and it is indeed a behaviour of the system. The counter example is a proven error, exit the test and show the result.

8. If the learner and model checker could not find another test string and random input generator has been used less than random queries number: use a random input in step 1 of length random string length.

9. No new states can be found, and no errors in the requirements have been found, exit test and show the results.
Chapter 5

Case study

The complete software (i.e system under test or SUT) had roughly 32000 lines of code and more than 800 inputs, while controlling and monitoring numerous functions of the aircraft. A critical safety requirement was chosen that took input from the wheels of the aeroplane and would determine and output what mode the aircraft was in. Different modes being: IN_AIR, or on the ground states: LANDING, TAKING_OFF, MOVING and NOT_MOVING. The function is critical since other modules rely on knowing the correct state of the aircraft. For example the radar should never be turned on when on ground, and the landing gear should not be able to be retracted when on the ground.

5.1 Requirements analysis

Requirements for the function were available in several documents at different “levels”, a choice was made to test requirements at a higher level since they are available at the earlier stages of development and the low level requirements where not complete. These high level requirements describe what transitions the system should handle, and what events should trigger state changes. Some events are specified to need longer consistent input to infer a state change, some state transitions occur immediately and others take up to several seconds. From the textual requirements these events see table 5.1 and state changes see table 5.2 were identified.

The reason for some events needing to take longer consistent time to change state is due to the possibility of input "noise", erroneous data or simply due to natural causes. The wheels may for example bounce when the plane leaves the runway, or the wheels might slip momentarily when touching down.

<table>
<thead>
<tr>
<th>Event</th>
<th>Wheels</th>
<th>Speed</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>moving</td>
<td>−</td>
<td>High</td>
<td>Immediate</td>
</tr>
<tr>
<td>not_moving</td>
<td>−</td>
<td>Low</td>
<td>Long</td>
</tr>
<tr>
<td>taking_off/touching_down</td>
<td>Rear</td>
<td>−</td>
<td>Immediate</td>
</tr>
<tr>
<td>abort_taking_off/landing</td>
<td>Rear &amp; Main</td>
<td>−</td>
<td>Short</td>
</tr>
<tr>
<td>V(P) or W(p)</td>
<td>None</td>
<td>−</td>
<td>Short</td>
</tr>
</tbody>
</table>

Table 5.1: Events that should trigger state changes between aircraft modes. Short means less than one second, long means more than one second. immediate means change should happen at the next execution cycle.
### Table 5.2: Possible state transitions from the state LANDING (a subset of all requirements)

<table>
<thead>
<tr>
<th>State</th>
<th>Event</th>
<th>Resulting state</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LANDING</td>
<td>moving</td>
<td>LANDING</td>
<td>Speed only</td>
</tr>
<tr>
<td>LANDING</td>
<td>not_moving</td>
<td>LANDING</td>
<td>Speed only</td>
</tr>
<tr>
<td>LANDING</td>
<td>taking_off/touching_down</td>
<td>LANDING</td>
<td>Rear wheels only</td>
</tr>
<tr>
<td>LANDING</td>
<td>abort_taking_off/landing</td>
<td>MOVING/NOT_MOVING</td>
<td>Depending on wheel speed</td>
</tr>
<tr>
<td>LANDING</td>
<td>in_air</td>
<td>IN_AIR</td>
<td>All wheels only</td>
</tr>
</tbody>
</table>

![Figure 5.1: Possible state transitions used to derive the requirements (not a correct state chart).](image)

#### 5.2 Requirement analysis

24 safety requirements where specified from the textual specifications and diagrams at system level, 5 of which were made for the state LANDING *see table 5.3*, the other states have very similar LTL requirements and will not be listed.

<table>
<thead>
<tr>
<th>Nr</th>
<th>LTL Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>G(ostate = LANDING &amp; input = moving -&gt; X(ostate = LANDING))</td>
</tr>
<tr>
<td>2</td>
<td>G(ostate = LANDING &amp; input = not_moving -&gt; X(ostate = LANDING))</td>
</tr>
<tr>
<td>3</td>
<td>G(ostate = LANDING &amp; input = back_wheels -&gt; X(ostate = LANDING))</td>
</tr>
<tr>
<td>4</td>
<td>G(ostate = LANDING &amp; input = all_wheels -&gt; X((ostate = NOT_MOVING)</td>
</tr>
<tr>
<td>5</td>
<td>G(ostate = LANDING &amp; input = no_wheels -&gt; X(ostate = IN_AIR))</td>
</tr>
</tbody>
</table>

*Table 5.3: Events that should trigger state changes the aircraft mode*

Note that requirements 1, 2 and 3 could be written within the same requirement with OR operators in between the input values. Also more than one LTL requirement could be added in an LTL formula using the AND operator. This would have consequences however when the requirement is tested and fails, we would not know which of the formulas that failed the check.

Some events in this case did not define all input *see table 5.1* making it up to the reader to interpret whether the data is implicit or obvious or find the information in other available documents or discuss with the designer. When writing the events and the wrappers, questions where raised on what would happen if that data changed. For example if the wheel speed could change while in the air or not and what would happen to the aircraft state then?

Two alternative wrappers where written, one where the wheel speed was set to 0 while in the air and one with leaving it unchanged. Creating two different models. Which one is better is up for debate but abstracting away behaviour that is not specified (setting wheel speed to 0 while in the air) is in that case a choice by the tester and not the designer. Therefore only the latter version was used. An additional requirement was written that no change should happen if the wheel speed...
changes while in the air.

5.3 The SUT input and sensory data

The function or sub module that handled the state of the aircraft was a part of the software with interconnecting and surrounding modules. Input was therefore sent not directly to the sub module but through the same way the actual sensor data would move using the inputs of the larger system. This also increases confidence in that the chain of input to the sub module through other sub modules are correct.

For safety reasons sensor inputs have redundancy by using more than one sensor. Input from a wheels sensors indicated if a wheel was pressed down or not. The front wheel in addition had speed sensors monitoring the speed of the wheel.

Because of the high level requirements that did not care about how many sensors were used only on speed and number of wheels see table 5.1, the input was partitioned into only 3 inputs. A sensor and its redundant counterparts for a wheel were made into one input in the wrapper. Also the left and right main(rear) wheels where grouped together as only one input in the wrapper see fig 5.2 for an abstract view of how the wrapped SUT looks like to LBTest.

![Figure 5.2: Abstract view of the input and output of the sub module being tested.](image)

5.4 LBTest integration

The development process at SAAB for this system was highly automated. Regular testing where already made on the system and its sub modules and code was generated using model based development from a graphical programming environment. Since LBTest needs an executable that can take standard input and output and also be able to terminate, a manual build and executable was made containing the wrapper and the code.

5.5 The Wrapper

Communication between LBTest and the SUT has to be made using an executable that uses standard input and output. LBTest will also repeatedly terminate the executable each time it sends a new trace of input symbols to make sure that it has the initial state. An adapter or wrapper had to be made that would translate signals into input to the SUT and also handled the abstraction level of the behaviour model.

1. Sampling rate - how often to send and receive new data.
2. Translate symbolic values into input and output data.
void setWheelBack()
void setWheelRight()
void setWheelAll()
void setRoseWheelSpeed(int speed)
void printState()

// run for certain time
void RunCycles(int cycles)
{
    for(int i = 0 ; i < cycles; i++)
        SUT.Execute();
}

int main()
{
    string inputString;
    vector signals;
    char signal;

    while(true) // every LBTest line ex: a,b,c,d,e,f,g
    {
        // reset all inputs and reset SUT to initial state
        SUT.Initialize();
        SUT.RunCycles();
        SUT.printState();

        /* for every character in the line, store them in vector signals
        If any signal was read print delimiter "," */

        // If any signal was read go through vector applying signals
        for (int i = 0; i < signals.size(); i++)
        {
            if(signals[i] == ',')
            {
                // send event representing i to SUT and execute
                SUT.setRoseWheelSpeed((i%4));
                SUT.RunCycles(S_EVENT_TIME);
                else
                    /* handle other symbols */
                SUT.printState(0);
                /* if there are more signals in the signals vector append "," */
            }
        }
        return 0;
    }

Figure 5.3: Simplified wrapper pseudo-code.

3. Abstract or refine output data – what data to send back to LBTest.

The wrapper for this thesis was made to be modifiable to quickly and easily change the abstraction level see fig 5.3. Functions where made to execute the SUT for a certain amount of time with a given input. Other functions where made to set several parameters at once based on the high level requirements of the events that should trigger state transitions. Examples are setWheelSpeed() and setWheelsAll().

5.6 Sampling time or sequential abstraction

If time abstraction (or sampling) is used either by having a fixed interval in between observation points or having different events take different time. There may be some behaviour in between the observed points. Figure 8 below shows possible behaviour in between observation points O1 and O2 of an event. If an event takes more than 1 step or cycle we will only observe the behaviour at its endpoints. If we only want to make sure that an event does happen this does not matter. However if there needs to be a certain or minimal/maximal amount of time when the transition should occur, we may want to add functionality for some error checks in the wrapper.

Possible cases are:
A. State transition happens late – transition is observed as ok probably the expected behaviour.

B. State transition happens early – transition is observed as ok if requirement states that it should take the full amount of time for the transition to happen. Then this behaviour is incorrect.

C. State transition switches back and forth to start point – transition observed as no change at all - Possible unwanted behaviour.

D. Transition moves several steps before it is observed. - transition is observed as ok Possible unwanted behaviour.

E. (not visualized), no state change at all, will be observed the same as C

Transitions like these could be taken care of by the wrapper, if the time for the event should always be late and the events take equal amount of time, this change would be relatively easy. For more complex events and different times the wrapper would need to distinguish between different kinds of events and add more functionality to the wrapper.

5.7 Different wrappers

During the case study different wrappers where made to explore different options for the SUT and evaluate the process of designing and implementing wrappers.

**Refinement with output of speed from the wrapper** Since some events change the speed and some events do not affect it, effectively the last set input speed will be used during subsequent events. The SUT does not output the speed but the wrapper can be set to remember the last set input speed (high or low) and output it back to LBTest. This changes the appearance of the state transition diagram output of lbttest, and makes it possible to write requirements in new ways. A disadvantage is that the output is from the wrapper and not the SUT. see appendix A.

**Wait until first change wrapper** A wrapper was written to continue executing with the same input until there was a change (or a long time had passed) in the output change and if there is a change immediately sends back the resulting state to LBTest. The advantage of this wrapper is that it may help the learner find some hard to learn traces (many similar values in a row see fig 5.5). A disadvantage is that it will miss other traces for example a state change on alternating input signals with no change in output values.
Multi variable input In LBTest a symbol may represent many parallel inputs. This makes it important to write every combination of parallel inputs that needs to be tested as a symbolic value, and make the wrapper handle them differently. Another solution is to let LBTest handle more than one symbol at a time. Here combinatorial testing may be considered but will not be discussed much further in this report.

When LBTest is configured for multiple outputs it uses pairwise combinatorial testing meaning that it is not necessary to test all combinations of input, but only a subset. Complexity and execution time of the number of variables when testing all combinations would increase with 2 to the power of n inputs. A disadvantage is that LBTest may produce a lot of combinations.

Symbolic names may also be used to represent a large number of inputs instead of letting LBTest handle many inputs at the same time which may lead to state explosions and combinatorial explosions[13].

Check errors in-between observation points Wrapper that counts the number of transitions in-between observation points and sends a signal to LBTest if it detects unwanted behaviour. This solution may find behaviour that an abstraction might miss. For example: if between two observation points there is more than one transition of state, there has either been a "blank" see fig 5.5 in the output or there where more than one transition to get the output at the next observation point. In some cases we might want to catch these behaviours as errors (Assuming we only want clean transitions of one step in between points). Using a requirements with a signal "stable": G(stable = true) for catching it every time, or together with other requirements: G (<transition>... AND stable = true). Another check that can be done by the wrapper is to find if the transition was late or early as in fig 8 A and B.

5.8 Learning algorithms and hard to learn models

One limitation on using learning algorithms for automata is how easy the model is to learn. How much time do we need to effectively learn the complete model? Some systems are easier to learn than others[6]. As part of evaluating LBTest a small program was made that aimed to be “unlearnable” by LBTest. The program took as input the symbols 0 and 1 an output A, B and C. The program counted upwards alphabetically when receiving a 1. And reset to A when receiving a 0 see fig 5.5A.

To make the system harder to learn an internal counter was added to the program that would increase by one for every 1 that was sent when state was in C, while doing so it would still output C. after a certain number of times a new state X was sent as an output see see fig 5.5B.

When running LBTest on the program the time to learn the system increased when increasing the number of the internal counter. A learner algorithm will only test sequential inputs up until a certain range depending on the algorithm. To find hard to reach states like these we need to rely on the random input generator. Getting the same input several times in a row with a random input is increasingly harder with number of possible inputs and number of sequential inputs needed.

The solution to this may be the possibility to adjust how deep the learning algorithm should go to find new possible states. The algorithm can only define a state as a new state if there is a new output at the end of the trace. LBTests random input generator can be configured differently to accommodate this. The counterweight being that the whole learning process may take an increasingly longer time. Knowing if a system has many same value outputs in a row is an advantage to know if the settings for the random input generator should be increased. Another possible solution is using the wrapper; it can either abstract sequence of data into observation points or refine data with adding more output in the wrapper itself.
Figure 5.5: Example of a counter and a possible "hard to learn" state X
Chapter 6

Results

This chapter shows the results of running LBTest against the specified SUT requirements and some findings of the LBTest integration process.

6.1 LBTest results

The final wrapper in the study used a simple wrapper version and did not record the state change time of the events. Only the specified requirements time and expected result at the observation points where used for the events see table 5.1. All 24 safety requirements passed in LBTest. The test took less than a few minutes see fig 6.1.

![Figure 6.1: LBTest with 6 of the 24 requirements](image)
Tests were done at different times using different model checkers: BDD, BMC and learning
algorithms: IKL and L* where used, the random input generator was set to 400 random length
and 400 random queries which may be considered a lot at this level. Time wise there where not
any significant difference between the model checkers and learners used. This may be to the fact
that the learnt model state size did never grow larger than 8 states with this level of abstraction.

6.2 Testing environment

The Environment used when running LBTTest was on a thin client – on a virtual servers network
with several hosts and users. LBTTest and the SUT were run on a virtual Linux server on a 3.6
GHz 64 bit Xeon processor with 100gb of ram. The nature of a shared system makes the system
performance somewhat unpredictable and execution time is based on overall load and network use
with a lot of users. Therefore this report cannot use precise comparisons or statistical measurements
of the performance between runs of the tool LBTTest. The performance numbers above are used to
give some relation to the run time of the tool, the results and the system used.

The development process at Saab was not made for manual running of test cases together with
an external program like LBTTest. Much manual work needed to be done to build the wrapper and
find the relevant inputs from the SUT. Once these steps where made, running LBTTest again was
relatively simple.

6.3 Low level testing

Running LBTTest with no sampling rate (every events duration was set to only 1 execution step)
yields a much larger model, running LBTTest with increasingly larger settings up until random
string length = 1000 , random queries = 100 000 found the state machine to be of 393 states
and took 18h and 39 minutes to complete. Different settings and learner algorithms could make
this time change and a very high number of random queries where used to increase confidence on
finding all states. With this sampling rate and the behaviour of the system there will be a lot of
sequential states with the same output, making this model much harder to learn. There where no
requirements written at this level which would be considered low level testing.

6.4 Writing requirements

LTL has a number of operators that can be used when translating textual requirements into LTL.
This thesis made use mostly of the Global and Next operators for safety requirements Other
operators Like Future and Until where tested on small examples when learning LTL and the only
limit found was if there is a need to state that something should happen a certain times in a row the
LTL requirements will be harder to read and write. G(expression = true AND N(expression =
true AND N(expression = true AND N......))) Finding relevant requirements in the existing
requirements for the System Under Test (SUT) at Saab and how they translate to LTL.

Writing small example programs is a good way of testing LTL requirements for a novice in the
tool. On small examples the graphical representation and the log files are easier to understand.
Creating a system that purposely is hard to learn to test what kinds of configuration of LBTTest
should be needed is also helpful for practical understanding of how the tool behaves.

Since the textual requirements for the SUT were easy to understand the writing of the LTL
came naturally and was done in less than a day. However much time was spent on discussions on
undefined behaviour like changing wheel speed while in air and making sure that the translated
requirements were done right. LTL had useful operators for the kind of high level requirements
used in this thesis. However they required a wrapper to accommodate the same level of abstraction as the requirements themselves.

6.5 Writing wrappers

Writing a wrapper is done using relatively simple programming, in this case the first version was done in less than a day. Most time related to the wrapper was made on figuring out the appropriate abstraction level, and testing different versions. During the writing of the wrapper a programming error was made which made some symbols exhibit the wrong behaviour. This made LBTest report errors in the requirements, finding the source of the problem whether it was the requirement, the SUT or the wrapper took some time.

Wrappers have a lot of potential in simplifying the model, and add additional behaviour to it. The adapter will be very simple for some projects and more advanced for others. What requirements need to be tested, the structure of the program, and how large the model will be each determine the need for abstraction and functionality in the wrapper.

Since wrappers very much determine how abstract the model representation will be, care must be taken to not abstract away observable information that is needed for the verification of the requirements. Still abstraction also makes certain requirements easier for LBTest to test and writing LTL requirements may be simpler. The requirements and events that where going to be tested for the SUT was on a higher level, therefore the wrapper needed to abstract away...
Chapter 7
Discussion

This chapter discusses the results of the study in regards to integrating LBTest, The LBTest algorithm and wrappers, as well as previous case studies.

7.1 Integrating LBTest

Integrating LBTest into the development process into a SUT is relatively easy. So is the implementation of a wrapper. Most work related to integration was understanding the development process at Saab. When the method of integration was found the process was repeatable but needed some manual work regarding compiling and running the SUT. LBTest may be implemented into the development process to make this step easier.

Setting up one LBTest configuration for complete testing of a SUT is hard and impractical, its seems better to manage several config files and tests related to the test level and grouping of tested functionality. When the Tests are configured rerunning them is simple.

LBTest may be used to provide quick feedback on requirements to the system designer and the developers. LBTest may be suitable for agile development and test driven development where focus is on testing the requirements from the beginning. The creation of the formal specifications also made discussions and an inspection of the functional and non functional requirements of the system. Proof of correctness has to consider how hard the model is to learn and on when LBTest is configured to expect the complete model to be learned. The state space explosion problem may not be the main concern when using LBTest, it’s a matter of defining what you want to verify then move on to write the requirements and then the wrapper abstraction of the system. In this case due to abstraction the state size for an industrial and large system became quite small.

7.2 LTL and Model Checking

Since LTL is an extension of propositional logic it is relatively easy to understand and write for those used to propositional logic or perhaps programming. Rewriting requirements into LTL formulas makes the writer review the specification and no ambiguity should be possible. One disadvantage found with the LTL specification or perhaps the model checker was that a missing output symbol(symbol is defined in the LTL requirements but was never recorded by the wrapper) would mean that all its related requirements would pass since no counterexample would exist. This is probably a bug with NuSMV. A manual check in the dot file or a script searching that the values exists at all may be done to counter this and perhaps LBTest could add a feature to warn when a specified input or output signal was never observed. There is a possibility that an LTL
requirement could be written that would fail if no such data output was observed. No such LTL requirement was found however.

LBTest will for each requirement restart the process of creating a model if this was not the case verifying the requirements after a model was found could be done much faster. This can be done manually using NuSMV directly on the last learnt model outside of LBTest. Being able to configure LBTest to use an old model for verifying requirements could be a helpful setting. An advantage of restarting the model creation process could be that since counter examples are used when creating the hypothesis model larger models may find errors quicker this way.

Since LBTest outputs its last hypothesis as a NuSMV file, one may manually use NuSMV to run additional requirements against the model manually outside of LBTest.

7.3 Learning algorithms

The limiting factor on learning algorithms and their ability to learn some systems easier than others makes it preferable if the user has some knowledge when a system is hard to learn and when it is not. Arguably this may be best done by the developer or anyone that has an understanding of the implementation. Determining how easy a system is to learn is often not an easy question to answer.

7.4 Wrappers

Many things can be done in wrappers and they seem to largely affect the size of the behavioural model of a system. Most importantly care must be taken as to not abstract away behaviour that we want to test. Also knowing that the only truth about the system is what is observed after the wrapper has done its job. Writing the wrapper is relatively simple with knowledge of LBTest’s communication protocol” LBTest also has error output when the wrapper does not communicate as expected and since the wrapper uses standard input and output it is easy to manually test it. The abstraction level of the wrapper was a bit forced by the sampling rate of the system and the requirements that was being tested.

It is possible that the most use of LBTest is when the user has some basic knowledge of LBTests inner workings such as model checkers, the random input generator and learning algorithms. Since LBTest is a platform new techniques can be implemented in the future and the possibility to adapt and change settings depending on the solution may be LBTest biggest advantage. The solution proposed after this project would be to start small with an abstraction and settings of LBTest that would run fast on the SUT to quickly find errors. When no errors are found the settings and even the wrapper/adapter could be modified to provide further proof. Using several wrappers and LBTest configuration files for different kinds of tests on the SUT is also a valid method.

Because of the many available LBTest configuration options, wrapper implementations and SUTs experience using LBTest also seems relevant. While knowledge of inner workings of LBTest is important there are general guidelines from the developers of LBTest of how to use the program and what settings to use first[18].

Larger model tests To make a larger model that LBTest could work with the sample rate could be changed to observe every clock cycle. However this may lead to a harder to learn model, and the random input generator may have to do almost the entire job and could perhaps be slower than only using random input. The model would be hard to learn since so many identical inputs would be needed in a row for an output change to occur. Also it will be increasingly unlikely for the random input string to contain such a combination.
State space size is certainly important to test and LBTest has been used effectively in several other case studies without state explosion see previous case studies 7.5. State size is also a research topic when it comes to model checkers[1]. However the learning algorithms limits and the process of writing wrappers and translating requirements to LTL at different levels could also be an interesting part of LBTest to explore further.

7.5 Previous case studies

Several case studies have been made and many if not all have provided feedback to the LBTest project. The Tool has changed between the case studies and during this study alone new features where quickly implemented when needed.

Summaries: The paper: Case studies in Learning Based Testing[2] summarizes the basic principles of learning based testing(LBT). The paper also mentions that learning based testing outperforms random input testing. There are two industrial case studies in the paper: "The Fredhooper access Server (FAS) a concurrent object oriented system in e-Commerce", and "An embedded Break By Wire system developed by Volvo Technology AB". In Both these case studies LBTest found requirement errors.

Case study: Learning-based Testing of a Large Scale Django Web Application[17] Provides a case study from the financial sector. The case study uses a technique for automatic database rollbacks to minimize the downtime between queries in the database environment when restarting the SUT. The case study found errors in the implementation of the SUT during the process of writing LTL requirements.

Learning-based Testing of Distributed Micro service Architectures[16]: The study shows results using LBTest with hardware and software fault injections, restarting services communication faults and killing service instances at different times.

7.6 Summary

The LBTest tool and learning based testing is an interesting method with much flexibility in regards of what requirements should be tested during different development stages. The tool may be configured for quick runs to find errors early, or configured for running extended periods to increase confidence at later stages. The Tool has provided usefulness in the case study for understanding the SUT from a requirements based perspective. Using LBTest forces the SUT requirements to be well written, unambiguous and thorough. The tool requires the user to have some theoretical knowledge of learning based testing and formal methods.

Since the tool is flexible and configurable it is hard to test all of its possibilities in one case study, and its usefulness may improve with the experience of the user. Learning based testing seems promising and may increase the effectiveness and confidence of testing. Development costs may be decreased due to early testing and the Tool LBTest may be a useful complement to traditional software testing.

Further studies may focus on low level testing and try to reach larger state space problems.
Appendices
Figure 1: 
A: Example of wrapper output refinement (added speed as output) 
B: Example of multiple inputs (combinatorial)
Bibliography


[17] S Lundmark 2013 Learning-based Testing of a Large Scale Django Web Application: An Exploratory Case Study Using LBTest KTH, School of Computer Science and Communication (CSC)