Model-Based Testing of Patient Ventilator

HADI MOVAGHGHAR
ABDULLAH AL AHAD

Master’s Degree Project
Stockholm, Sweden March 31 2014

XR-EE-LCN 2014:003
Model-Based Testing of Patient Ventilator

Masters’ of Science Thesis
Department of Information and Communications Systems Security
Stockholm, Sweden

Hadi Movaghghar
Abdullah Al Ahad

Supervisors
Farshid Bagherpour
Leif Lychou

Examiner
Panagiotis Papadimitratos

Stockholm, March 2014
Abstract

Software testing is a very significant part of Software Development Life Cycle. The classification of software testing has become pretty large and IT firms are implementing one or a combination of testing methodologies to ensure the quality of their software products. More coverage and less-repetition of test scenarios are two significant issues in software testing. Lack of effective automation makes the process repetitive, error-prone, time-consuming and costly. Therefore, the automation of test procedure has become a vital issue in software industry.

As the options and capabilities in patient ventilation systems are increasing day by day, so are going for the number of parameters’, their values and ranges. So that it has become impossible to test all input parameters and their dependencies on manual testing procedures.

From the industrial point of view, Model-Based Testing (MBT) is a prospective technique to deal with these issues by algorithmic generation of test cases automatically from a model of requirement analysis. The objective of this thesis is to perform MBT of application software in patient ventilators so that bug area of test scenarios can come under consideration with the automation of testing procedure. In this thesis work, parameters in different categories of patient ventilation are analyzed and a model is made with high priority parameters. A user interface is made for the ease of selecting parameters’ ranges and different options. Combinatorics generates unique combinations of input values. The model is used to generate large amount test cases automatically. These test cases are sent to System Under Test (SUT) or target systems through an adapter for execution. The purpose of the adapter is to integrate the model into company’s existing test framework. Finally, by analyzing the output of the target system, model and real scenarios further improvement is done for model in iterative fashion for Quality Assurance (QA).

Keywords:
Model-based testing (MBT), Testing and verification, Testing automation, Patient ventilators
Acknowledgement

For completing our masters’ thesis we are grateful to many people in the organization MAQUET Critical Care AB and to our educational institute The Royal Institution of Technology. Without relentless help and support of them, it would not have been possible for us to implement the theoretical idea of this thesis and write the whole report. We would graciously like to express our appreciation to them for inspiring and motivating us throughout the whole work duration. We would like to take this opportunity to thank few of them who we believe deserve their name to be mentioned here.

We would like to start by mentioning the name of the people in the organization. First of all, we would like to thank our two supervisors, Farshid Bagherpour who is the Test Design Engineer and Leif Lychou who is the System Architect Engineer of the organization. Though they were very busy people with lots of responsibilities in the organization, we never felt deficiency when it came with the matter of helping us in providing technological infrastructure and theoretical knowledge. Their insightful discussions, close supervision and expert comments helped us to deliver a quality output at the end of the day. We would like to thank our manager Anders Östberg for giving a substantial effort to provide us a helpful working environment and resource when it was necessary. We would also like to thank two other people in the organization. They are Nikos Anastasiadis and Pawel Defee.

In our educational institution, firstly we are grateful to our professor Panos Papadimitratos for his insightful sessions with us. Those sessions were significant in providing guidance to follow the path of the implementation and writing report. For his vast experience of research fields, it became easy for us to see the way where it would probably not be possible without his knowledge sharing. We particularly want to express our gratitude to him for his patience to us for completing the whole work. In addition, we would like to spread our area of gratitude to those professors at KTH who helped us to build our concept throughout the whole masters’ program.

Last of all, we would like to remember our families who kept continuing their mental and financial support through our studies and this research. Without their support, completion of a masters’ degree in a country like Sweden would remain dream to us. We would also like to acknowledge them for their great contribution.

Best Regards,

Hadi Movaghghar
Abdullah Al Ahad
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AETG</td>
<td>Automatic Efficient Test Generator</td>
</tr>
<tr>
<td>API</td>
<td>Application Platform Interface</td>
</tr>
<tr>
<td>ASML</td>
<td>Abstract State Machine Language</td>
</tr>
<tr>
<td>AVA</td>
<td>Advanced Ventilation</td>
</tr>
<tr>
<td>CPAP</td>
<td>Continuous Positive Airway Pressure</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheets</td>
</tr>
<tr>
<td>EFSM</td>
<td>Extended Finite State Automata</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Automata</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HEVT</td>
<td>High End Ventilation Tool</td>
</tr>
<tr>
<td>I:E</td>
<td>Inspiration Expiration Ratio</td>
</tr>
<tr>
<td>MBT</td>
<td>Model Based Testing</td>
</tr>
<tr>
<td>MBTUC</td>
<td>Model Based Testing User Conference</td>
</tr>
<tr>
<td>MV</td>
<td>Minute Volume</td>
</tr>
<tr>
<td>NAVA</td>
<td>Neurally Adjusted Ventilatory Assist</td>
</tr>
<tr>
<td>NIV</td>
<td>Non Invasive</td>
</tr>
<tr>
<td>PC</td>
<td>Pressure Control</td>
</tr>
<tr>
<td>PEEP</td>
<td>Positive End Expiratory Pressure</td>
</tr>
<tr>
<td>PRVC</td>
<td>Pressure Regulated Volume Control</td>
</tr>
<tr>
<td>QA</td>
<td>Quality Assurance</td>
</tr>
<tr>
<td>QML</td>
<td>Conformiq Modeling Language</td>
</tr>
<tr>
<td>RCP</td>
<td>Rich Client Perform</td>
</tr>
<tr>
<td>RR</td>
<td>Respiratory Rate</td>
</tr>
<tr>
<td>SIMV</td>
<td>Synchronized Intermittent Mandatory Ventilation</td>
</tr>
<tr>
<td>SUT</td>
<td>System/Software Under Test</td>
</tr>
<tr>
<td>TMT</td>
<td>Test Model Toolkit</td>
</tr>
<tr>
<td>TPT</td>
<td>Time Partition Testing</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>VC</td>
<td>Volume Control</td>
</tr>
<tr>
<td>VT</td>
<td>Tidal Volume</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
# Table of Contents

## 1 INTRODUCTION

1.1 ORGANIZATION ................................................................. 1
1.2 BACKGROUND ........................................................................... 1
1.3 PROBLEM STATEMENT .............................................................. 3
1.4 PURPOSE ..................................................................................... 3
1.5 GOAL ............................................................................................ 3
1.6 CHALLENGES AND REQUIREMENTS .............................................. 3
1.7 METHODOLOGY .......................................................................... 4
1.8 CONTRIBUTIONS OF THE THESIS ............................................... 4
1.9 THESIS OUTLINE ................................................................. 5

## 2 BACKGROUND

2.1 SOFTWARE TESTING TERMINOLOGIES ........................................ 6
2.2 TEST AUTOMATION ..................................................................... 7
2.3 MODEL-BASED TESTING ............................................................ 8
  2.3.1 MBT Approaches ................................................................. 8
  2.3.2 MBT Steps ........................................................................... 9
  2.3.3 MBT Classification ............................................................... 11
2.4 BASICS OF COMBINATORICS ................................................... 13
2.5 BASICS OF PATIENT VENTILATOR ............................................ 14
  2.5.1 Invasive .............................................................................. 15
  2.5.2 Non-Invasive Ventilation (NIV) ........................................... 17

## 3 RELATED RESEARCH, REAL WORLD IMPLEMENTATION AND MBT USER SURVEY

3.1 RELATED RESEARCH .............................................................. 18
3.2 REAL WORLD IMPLEMENTATION ............................................. 19
  3.2.1 Model-Based Testing at Microsoft .................................... 19
  3.2.2 Model-Based Testing in Embedded System ....................... 19
  3.2.3 Model-Based Testing in Automotive Industry .................. 20
3.3 MBT USER SURVEY ............................................................. 20

## 4 DESIGN DESCRIPTION .......................................................... 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>APPLICATION OVERVIEW</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>APPLICATION ARCHITECTURE</td>
<td>22</td>
</tr>
<tr>
<td>4.3</td>
<td>APPLICATION IO</td>
<td>23</td>
</tr>
<tr>
<td>4.4</td>
<td>GUI</td>
<td>23</td>
</tr>
<tr>
<td>4.5</td>
<td>APPLICATION ENGINE</td>
<td>23</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Combinatorics</td>
<td>24</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Model</td>
<td>24</td>
</tr>
<tr>
<td>4.6</td>
<td>ADAPTER</td>
<td>24</td>
</tr>
<tr>
<td>4.7</td>
<td>UNDERLYING SYSTEMS</td>
<td>24</td>
</tr>
<tr>
<td>4.8</td>
<td>CHOOSING TOOL</td>
<td>25</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Choosing MBT Tool</td>
<td>25</td>
</tr>
<tr>
<td>4.8.2</td>
<td>Requirement:</td>
<td>25</td>
</tr>
<tr>
<td>4.8.3</td>
<td>Commercial Tools</td>
<td>26</td>
</tr>
<tr>
<td>4.8.4</td>
<td>Open-source Tools</td>
<td>27</td>
</tr>
<tr>
<td>4.8.5</td>
<td>Choosing platform</td>
<td>28</td>
</tr>
<tr>
<td>5</td>
<td>IMPLEMENTATION</td>
<td>29</td>
</tr>
<tr>
<td>5.1</td>
<td>OVERVIEW</td>
<td>29</td>
</tr>
<tr>
<td>5.2</td>
<td>CODING IMPLEMENTATION</td>
<td>31</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Domain package</td>
<td>31</td>
</tr>
<tr>
<td>5.2.2</td>
<td>GUI</td>
<td>31</td>
</tr>
<tr>
<td>5.2.3</td>
<td>GUI resources</td>
<td>33</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Application data and state</td>
<td>33</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Combinatoric</td>
<td>34</td>
</tr>
<tr>
<td>5.2.6</td>
<td>Model</td>
<td>36</td>
</tr>
<tr>
<td>5.2.7</td>
<td>Application IO system</td>
<td>38</td>
</tr>
<tr>
<td>5.2.8</td>
<td>Workspace structure</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>Schemas, validation</td>
<td>38</td>
</tr>
<tr>
<td>5.2.9</td>
<td>XML handlers</td>
<td>39</td>
</tr>
<tr>
<td>5.2.10</td>
<td>Adapter</td>
<td>43</td>
</tr>
<tr>
<td>5.2.11</td>
<td>Adapter scripts</td>
<td>43</td>
</tr>
<tr>
<td>5.2.12</td>
<td>Adapter in detail</td>
<td>44</td>
</tr>
<tr>
<td>5.2.13</td>
<td>Adapter utilities</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>Underlying systems</td>
<td>46</td>
</tr>
<tr>
<td>5.2.14</td>
<td>Deployment</td>
<td>47</td>
</tr>
</tbody>
</table>
6 QUALITY ASSURANCE

6.1 GUI TESTING

6.2 MODEL AND ADAPTER TESTING

   6.2.1 Phase 1: Tabular Representation (Limit Value)
   6.2.2 Phase 1: Graphical Representation (Limit value)
   6.2.3 Phase 2: Tabular Representation (Limit Value)
   6.2.4 Phase 2: Graphical Representation (Limit Value)
   6.2.5 Phase 3: Tabular Representation (Random Value)
   6.2.6 Phase 3: Graphical Representation (Random Value)

7 LIMITATION AND FUTURE WORK

8 CONCLUSION

9 BIBLIOGRAPHY
Table of Figures

Figure 2.1: Components of a test automation framework [10].................................................................7
Figure 2.2 Specific Activities of Model Based Testing Approach [12].......................................................10
Figure 2.3: Difference between Offline and Online MBT .................................................................12
Figure 3.1: Overall effectiveness of MBT ..............................................................................................20
Figure 3.2: Comparison efficiency between MBT and manual testing [26] ..............................................21
Figure 4.1: Application design in layer ..................................................................................................22
Figure 5.1: Iteration of MBT Application through Test suits.................................................................29
Figure 5.2: Detailed view of Check test case action state of figure .......................................................30
Figure 5.3: Test suite XML sample ........................................................................................................40
Figure 5.4: A portion of XML Output by Adapter .....................................................................................42
Figure 5.5: A portion of XML Output by Model .....................................................................................42
Figure 5.6: Workflow of Adapter for a single test case .........................................................................44
1 Introduction

This masters’ thesis work is conducted at a Swedish company named MAQUET Critical Care AB. It is a full time work according to regular Swedish office hours. Two supervisors were responsible for monitoring the activities in timely manner and two reports are submitted to them at the end of the thesis. One examiner is responsible from KTH side to grade this work.

1.1 Organization

MAQUET is a subsidiary of the publicly-listed Swedish group of companies GETINGE AB. The MAQUET brand represents the Medical Systems business division. It is a trusted partner for hospitals and clinicians since 1838. Maquet is a global leader in medical systems that advance surgical interventions, cardiovascular procedures and critical care. Maquet develops and designs innovative products and therapeutic applications to improve outcomes and quality of life for patients.

1.2 Background

Testing has become very important part of software development particularly for life-critical, financial transaction, real-time and embedded systems. These systems need huge amount of testing to make sure the 100% expected behavior for every situation for the long-term. Besides testing parameters, security measurements have also become necessary. The amount of data for testing is exponentially increasing day by day. To cover every input-output combination of every test scenario with different manual testing is becoming impossible day by day. On the other side not a single scenario is safe to skip for those systems. For example, the 1997 Mars Pathfinder mission began experiencing system resets at seemingly unpredictable times soon after it landed and began collecting data. Fortunately, engineers were able to deduce and correct the problem, which occurred only when (1) only one particular type of data was being collected and (2) intermediate priority tasks exceeded a certain load, allowing a blocking condition that eventually triggered a reset [1].

Even before the Pathfinder incident, it is shown by NASA researchers that the probability of fault density (number of faults per line of code) in rarely executed code is 100 times more than frequently executed code in a part of a program [2] [3]. In a 1999 study that considered faults arising from rare conditions, NIST reviewed 15 years of medical device recall data, in an effort to determine what types
of testing could detect the reported faults [4]. For example, one recall report indicated that the “upper limit CO2 alarm can be manually set above upper limit without alarm sounding.” In this case, a single parameter – CO2 alarm value – caused the problem, and a test with the upper limit value exceeded could have detected it. Another report gave an example of a problem triggered only when two conditions were met simultaneously: “the ventilator could fail when the altitude adjustment feature was set on 0 meters and the total flow volume was set at a delivery rate of less than 2.2 liters per minute”. In this case, a test in which the pair of conditions was true – altitude is 0 and rate is less than 2.2 liter/minute – could have detected the flaw.

In compare with Pathfinder program that consists of 155,000 lines of code (operating system code not included), commercial software like Boeing 777 airliner flies on 6.5 million lines of code, Microsoft Windows XP operating system is around 40 million lines of code. It is predicted that within next two years, average new car may have more than 100 million lines of code in various subsystem. Ensuring correct operation of complex software is so difficult that more than half of a software development budget – frequently tens of millions of dollars – is normally devoted to testing, and even then errors often escape detection. In 2002, Research Triangle Institute (a NIST-funded study) conducted surveys with both software and the national and found annual costs of an inadequate infrastructure for software testing is estimated to range from $22.2 to $59.5 billion [5].

Patients' ventilators have different ventilation systems where each mode has specific or global parameters depending on patient categories. For those options as input the system has an interface (touch screen interface) with different options. Neurally Adjusted Ventilatory Assist is an example of the latest breakthrough in mechanical ventilation. It requires some testing in the form of Exploratory or Session-Based testing in order to be able to cover these combinations as subsystems or system test.

It is not wise to make a big telecommunication system, embedded system or some kinds of automated system without having at least an abstract idea about their expected behavior. The feasibility test is needed before providing technical effort to build the real system. Otherwise after building the system if it does not fulfill the expectation then the effort will be in vain. For this reason the Model-Based test is used where modules and their relations are built abstractly (called Model) and black box testing is performed in that abstract model to assure the behavior. As with any engineered system, cost is a critical issue for quality software. Any small development in software testing area can provide a significant impact where testing consumes more than half of the development budget. Therefore, the necessity of automation, implementation of different algorithms, using of mathematical formulas for input-output combinations gained popularity in this area.
1.3 Problem Statement

Today the ventilation system has come up with lots of options. The option is so many that it is insufficient to get a satisfactory coverage with Exploratory or Session-Based testing [6]. This depends, first and foremost, that it is time-consuming to write test cases for all test combinations that must be run under the system and subsystem test. And there is no way to verify if all test combinations are actually tested or prevent the tests to test the same test scenario several times although Exploratory and Session-Based testing be used to increase test coverage.

Another way to tackle this is to use the Model-Based Testing (MBT), which define a model of the system, with all the associated parameters e.g. patient categories, ventilation system, etc. and how they depend on each other in the system with the help of Combinatorics methodology. The model can then be executed automatically by using the existing auto-test tools. MBT can help with obtaining a controlled testing of different and unique combinations of parameters or permission changes to the system without having to write test cases. MBT can first and foremost be used during stabilization period to find fault and get the test time for all permits.

1.4 Purpose

The aim of the project is to help the ventilation system to update to the next level by implementing the prototype and perform a comprehensive testing.

1.5 Goal

The goal of the project is to perform the Model-Based testing to the patient ventilation system. To support main goal and to develop the project on time with desired results, the project is divided into three phases:

**Pre-study**: How MBT can be used in medicine technical products? Here ventilators will be chosen as example product

**Feasibility**: How this can be done?

**Implementation**: At this phase the implementation of the "prototype" will be performed and associated documentation will be written.

1.6 Challenges and Requirements

Following are the challenges and requirements of this thesis work:
• Turning into the Maquet patient ventilator and be able to model the product at a high level
• Perform feasibility study on what is appropriate and feasible for Maquet's use of MBT methodology
• Find a suitable Combinatorics or other relevant device if necessary
• Model a number of different models of patient categories, ventilation modes and other important parameters
• Develop GUI tools to develop models and execute model easier
• How MBT can be used for both host and target environment
• How MBT test tool can be integrated with existing automation test tools
• Building the test system as modular as possible, with well-documented interface
• Documentation problem, solution and design choice to maintain and handover to the company for future extension of this work

1.7 Methodology

For doing this project inductive approach is taken. Empirical study of the current ventilator system and current testing tools is done; furthermore, a literature review of Model-Based Testing, required tools and similar systems is conducted. After finding out how to model the system using MBT, the easier way to integrate with existing test tools and design a user-friendly GUI, the result was a designed artifact. The collection data was largely qualitative through literature and empirical study.

1.8 Contributions of the thesis

As our thesis was in an organization, most of the contributions are related to that place. It upgrades the existing test framework of the organization from manual environment to automated environment. With our application, now a developer can test the system without knowing the technical details of the testing procedure. So the organization does not need a dedicated group for software testing. They do not even need to know the existing test framework because a comprehensive GUI of the application will guide them. The GUI simulates the existing system’s GUI so the user will feel no difficulties or will not need to adapt any new situation. Rather than executing one particular scenario at a time now the user can run lots of scenarios overnight and check the result on the following day. In this way, they can cover a larger area then they used to do before.

In addition, this thesis contributes in the field of Model-Based Testing. Basically this thesis is an implementation of Model-Based testing for embedded system. So the implementation signifies the importance of MBT implementation in this area. We had a very important realization while
implementing our thesis which was the model is not only important for automate the testing of an embedded system but also important while a new version of the product need to be released. The new product can be less error prone if it uses model’s implementation. A product cannot be released every day but the model can be changed every day according to the requirement. With the continuous feedback and testing, the model becomes more stable and less erroneous day by day.

1.9 Thesis Outline

The rest of this report is organized as follows. The second chapter contains a comprehensive background of the topics that are related to this thesis. Chapter 3 is composed of MBT-related researches, real-world implementations and examples of the survey done by the end users. The design of our application is the topics of Chapter 4, followed by the implementation in Chapter 5. Our application testing result and quality assurance procedure is discussed in Chapter 6. Limitation and future work is written in Chapter 7 and the thesis is concluded with Chapter 8.
2 Background

To design the model of patient ventilator and implement MBT, it is important to have an idea about some related concepts. This chapter will begin with some definition of testing terminologies used in this paper. After that MBT, its classification and different MBT tools will be described. One or two examples of the usage of MBT in this chapter provide a comprehensive idea about the usage of MBT in industrial area. The discussion is followed by a basic idea of combinatorics. This chapter is finished by a high level summary of patient ventilator.

2.1 Software Testing Terminologies

There are some common terms used in software testing industries. While going through this thesis work, these terms were necessary to understand in the implementation phase:

**Black Box Testing:** It is the testing without knowledge of the internal workings of the item being tested. This kind of tests based on the behavior of the component or system, derived from a specification. It is also known as functional testing or behavioral testing [7].

**Unit Testing:** It is a method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine if they are fit for use [8].

**Test Case:** It is a smallest unit of testing consists of information such as requirements testing, test steps, verification steps, prerequisites, outputs, test environment, etc. Alternately, it is a set of inputs, execution preconditions, and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

**Test Suite:** A collection of tests used to validate the behavior of a product. Basically, a test suite contains one or more test cases for the software under test [9].

**Software Under Test (SUT):** The SUT is the software or system that we intend to test. The purpose of the test is check one or more characteristics of this software [10].
2.2 Test Automation

A test automation system requires different components depending on its type, such as test scripts, input data, test oracles, a test driver and a test harness. These basic components are illustrated in below figure:

![Test Automation Framework](image)

**Figure 2.1: Components of a test automation framework [11]**

Definitions of the components are given below:

**Test harness:** A test harness for a SUT has several roles including setting up the initial state of the SUT for each test and setting up the testing environment (SUT and interacting components). One basic function of a test harness is to isolate the unit under test from the rest of the system. This is typically done by using a set of test components referred to as test stubs. When these are made programmable, they are often referred to as mock objects.

**Test Script:** A short program written in a programming language is used to test part of the functionality of a software system. It is commonly used to refer to the automated test procedure used with a test harness.

**Test Oracle:** Oracle (in test environment) is a mechanism to produce the expected outcomes to compare with the expected outcomes of the SUT. Test Oracle is a basic form of a test case that provides some input to the SUT and observing the resulting output. The output then needs to be asserted in order to define if the result was correct and matches the expectations set for it. The component of a test automation system that does this is called the test oracle.

The oracle is divided into the oracle information, specifying what constitutes the correct behavior, and the oracle procedure, which is the algorithm verifying the test results against the oracle information.
A test driver controls the overall test execution process, including the execution of the SUT. The SUT is isolated, for different testing purposes, from parts of its environment (other components or systems with which it interacts) with the help of a test harness. Test input can take different forms, such as message sequences (test scripts) and data values. Generators can also be used to automatically generate large quantities of different types of data. A test oracle is used to verify the correctness of the received output (such as data values or message sequences) in relation to a given input. In the context of this thesis, the SUT is considered as a black box that takes some input and produces some output, with only limited insight into its internal processes [11].

2.3 Model-Based Testing

Model-based testing is used to automate the design of black-box testing. Testing is performed considering a model of the target system as an engine. As a base of this testing, requirement analysis, design specification are used rather than the code. Only abstraction of the target system is taken care of while making the model. It means that the low level details are omitted. In most cases, only a part of the whole system is considered as a target system which we call System Under Test (SUT). In that case, only that relevant part is modelled for test case generation. Then, concrete test cases are generated from the model using the analysis of all transitions of the model (Combinatorics is used for this, described later), test-generation algorithm and MBT tool.

2.3.1 MBT Approaches

The following are the four main approaches known as model-based testing [12].

1. Generation of test input data from a domain model
2. Generation of test cases from an environment model
3. Generation of test cases with oracles from a behavior model
4. Generation of test scripts from abstract tests

Brief description of the four approaches is given below:

When model-based testing is used to generate test input data, the model is the information about the areas of the input values and the test generation involves clever selection and combination of a subset of those values to produce test input data.

The second approach of model-based testing uses a different kind of model, which describes the expected environment of the SUT. For example, it might be a statistical model of the expected usage of the SUT (operation frequencies, data value distributions, etc.). From these environment models it is possible to generate sequences of calls to the SUT. However, like the previous approach, the generated sequences do not specify the expected outputs of the SUT.
The third meaning of model-based testing is the generation of executable test cases that include oracle information, such as the expected output values of the SUT, or some automated check on the actual output values to see if they are correct. To generate tests with oracles, the test generator must know enough about the expected behavior of the SUT to be able to predict or check the SUT output values.

The fourth meaning of model-based testing is quite different: it assumes that we are given a very abstract description of a test case, such as a UML sequence diagram or a sequence of high-level procedure calls, and it focuses on transforming that abstract test case into a low-level test script that is executable. With this approach, the model is the information about the structure and API (application programming interface) of the SUT and the details of how to transform a high-level call into executable test scripts.

After going through different approaches of MBT and our target system (patient ventilator) we came into the following conclusion:

Among those four kinds of approach, the third approach has comparably greater potential though it is little bit complex and challenging. It is challenging because the test generator must know enough about the expected behavior of the SUT to be able to predict or check the SUT output values. Our target system is already a fully operational product from a long time ago and we have contacts with the technical experts of this system. Besides, as the concrete equipment and simulator is available it is possible to collect as much data as we need to understand the behavior.

Another advantage of this approach is that it is the only one of the four that addresses the whole test design problem from choosing input values and generating sequences of operation calls to generating executable test cases that include verdict information.

The last and main reason is: It matches mostly with our thesis requirements. Rather than manually writing tests based on the requirements documentation, this approach create a model of the expected SUT behavior, which captures some of the requirements. Then the model-based testing tools are used to automatically generate tests from that model.

2.3.2 MBT Steps

The following figure illustrates the steps of the MBT in details:
A small description of the steps in Model-Based testing is given below:

**Build the model:** Forming a mental representation of the system’s functionality is a prerequisite to building a model for testing purposes. The model should be a depiction of the software’s behavior, which can be described in terms of the input sequences accepted by the system; the actions, conditions, and output logic; or the flow of data through the applications, modules, and routines.

**Generate expected inputs:** The second step is to use the model to generate test cases, which consist primarily of specifying the inputs and expected outputs. The difficulty of generating tests depends on the nature of the model. In the case of finite state machines, it is a matter of implementing an algorithm that randomly traverses the state transition diagram (a directed graph). Tests are, by definition, the sequence of inputs along the generated paths. Thus, if the model is well defined, the tests can be generated automatically. In contrast, without automation and modeling tools, this task can be immense and near impossible to do manually for a complex system.

**Generate expected outputs:** Software testing involves execution of a program under test using some fault-revealing input data and examination of the output to determine success or failure. A fundamental assumption of this testing is that there is some mechanism, a test “oracle”, which is described previously. As illustrated in the summary Figure above, expected outputs must be generated prior to running tests. A test oracle is the criterion used to check the correctness of the output.
Run tests: Most MBT environments are supported with test generation tools that generate tests (test cases) that can easily be translated into executable test scripts, or produce the test script directly from the test data contained within the tool.

Compare actual with expected outputs: The automation process in place should make the comparison of actual to expected outputs, and alert testers to the failures.

We get different kinds of feedback like missing requirements, incorrect input/output, ambiguous and incomplete scenario etc. after executing the test cases. This kind of feedback and information again can be used to improve the model. It is an evolutionary process.

2.3.3 MBT Classification

MBT has a large taxonomy. In paper [14], author described the taxonomy of MBT based on different criteria. In addition, in document [15], two important classification of model-based testing is described. They are:

Offline MBT: Offline MBT helps to automate test case generation with providing the opportunity to execute these test case independently. It means Offline MBT generates a finite set of tests and execute those later. This allows automatic test execution in third party test execution platform.

Online MBT: In this case, Test case generation and execution are in motion. The next step is design after the output receiving. It is used for testing nondeterministic systems. It is possible to run infinite test suite in this kind of MBT.

The figure below illustrates the difference between these two kinds of MBT:
Following two classifications are studied by us thoroughly because the most tools are made on the basis of this.

**Transition Based MBT:** Transition Based MBT is performed on those systems which contains lots of states with comparably less amount of data for transition between two states. For example, if we think about bringing a complex telecommunication protocol under the review of model-based testing it will fall under transition based MBT. A simple telephone call establishing with H.323 protocol (a VoIP protocol) takes 10 steps from the beginning to the end [16].

**Data Based MBT:** Data Based MBT is performed on those systems which comparably has less amount of transitions but for one transition lots amount of data and calculations need to be performed. For example, patient ventilator has only 3 or 4 steps like initial, standby, running or active, alarm. However, for the transition from standby to running state all the parameters' value needs to set and huge calculation is necessary for selecting the parameter limit, alarm limit, output, output limit, etc.
2.4 Basics of Combinatorics

From the word COMBINATORICS it can be easily deduced that it is the science of combinations. Combinatorics is the branch of mathematics where usually the finite families of sets are studied. Certain characteristic arrangements of their elements or subsets are given, and asked what combinations are possible, and how many there are [17].

Now-a-days Combinatorics has become a significant part of mathematics and computer science. The reason is it has made a great impact in the speed of mathematical calculation. Therefore, currently computers have been able to solve large-scale problems that previously would not have been possible. But for making computers able to do this they need to be programmed with combinatorial algorithms and for making these algorithms more efficient (for reducing time and space complexity) more combinatorial thinking is necessary. Below the definition of Combinatorics is given in details:

Combinatorics is concerned with arrangements of the objects of a set into patterns satisfying specified rules. In case of combination and permutation for an arrangement two general types of problems occur repeatedly:

• **Existence of the arrangement.** If one wants to arrange the objects of a set so that certain conditions are fulfilled, it may not be at all obvious whether such an arrangement is possible. This is the most basic of the questions. If the arrangement is not always possible, it is then appropriate to ask under what conditions, both necessary and sufficient, the desired arrangement can be achieved.

• **Enumeration or classification of the arrangements.** If a specified arrangement is possible, there may be several ways of achieving it. If so, one may want to count or to classify them into types.

Two other combinatorial problems often occur.

• **Study of a known arrangement.** After one has done the (possibly difficult) work of constructing an arrangement satisfying certain specified conditions, its properties and structure can then be investigated.

• **Construction of an optimal arrangement.** If more than one arrangement is possible, one may want to determine an arrangement that satisfies some optimality criterion-that is, to find a "best" or "optimal" arrangement in some prescribed sense.

Thus, a general description of combinatorics might be that combinatorics is concerned with the existence, enumeration, analysis, and optimization of discrete structures [18].

Our system contains a numerous parameters and those parameters have different ranges. To deal with those parameters we need the concept of Combinatorics. For example, if we want two parameters to be
fixed and four parameters to vary within their certain ranges, we need to generate all the possible combinations without repetition. In addition, there are some other constrains that we may need to put on some parameters and some may be not. For simulating those particular scenarios it is necessary for us to find a good Combinatorics tool which will not only make all possible combinations but also allow us to provide constrains more dynamically.

2.5 Basics of Patient ventilator

An automatic breathing system known as patient ventilator which is needed when a patient cannot ventilate/breathe adequately, resulting gas exchange. The system supports three categories of patients. They are: Adult, Pediatric and Neonate. For every patient category, there are different ventilation modes to control the ventilation. To represent all the functions of a ventilator, electrical signals in the Patient Unit are used. Besides providing the options for setting parameters, modes and categories, the user interface also displays a continuous and real-time recording of the inspiratory and expiratory pressure, flow and volume. A full respiratory cycle comprises of three phases. They are: Inspiration, Inspiratory pause, Expiration. These phases are controlled by different parameters in a particular ventilation mode. The important parameters are:

- **Positive End Expiratory Pressure (PEEP):** It is the continuous remaining pressure maintained by patient ventilator inside the lung that prevent the walls of lungs to touch with each other.
- **Tidal volume (VT):** Volume (of air) per breath or target volume. The unit is milliliter (ml)
- **Respiratory Rate (RR):** Rate of controlled mandatory breaths. Used for calculating target volume. The unit is breath per minute (b/min).
- **Minute volume (V_{min}):** Volume per minute or target Minute volume (ml/min or l/min).
- **PC above PEEP:** Inspiratory pressure level for each breath in Pressure Control ventilation mode. The unit is centimeter/water pressure (cmH2O).
- **PS above PEEP:** Inspiratory pressure support level for triggered breaths (cmH2O) in Pressure Support.
- **Inspiratory rise time (T_{inspiratory rise}):** Time to full inspiratory flow or pressure at the start of each breath, as a percentage of the breath cycle time (%) or in seconds (s).
- **Inspiration time (T_i):** Time for active flow or pressure delivery to the patient.
- **I:E ratio (I:E):** Ratio of Inspiration time + Pause time to Expiration time.
- **Pause time (T_{pause}):** Time for no flow or pressure delivery (% or s).

In addition, there are other parameters, different ventilation modes, startup configuration values described in the user manual [19]. For modeling the system, important parameters are chosen.
Importance are determined by the amount of dependencies of a parameter to the other parameters. These dependencies are presented by mathematical equations in design document. The system has also parameters for different alarms settings. But for modeling those are not taken into consideration because of time limit.

Volume Control, Pressure Control, Pressure Support and Volume Support are the basic ventilation modes. The term basic is used for these modes because of the fact that these modes are the cornerstones in all of the other modes. PRVC for example is only a combination of VC and PC, SIMV and AutoMode are also combinations of the basic modes.

The composite modes name are: PRVC, Bivent, SIMV(VC), SIMV(PC), SIMV(PRVC), Automode(VC-VS), Automode(PCPS), Automode(PRVC-VS) and NAVA. A composite mode is a mode with the ability to control other modes. This makes it possible to make combinations of the basic modes to get more advanced modes, and thereby reusing the basic modes. PRVC for example is only a combination of VC and PC. It has a VC start breath and the other breaths are PC breaths with the pressure adapted at each breath.

Basically a patient ventilator has two types of ventilation. They are: Invasive and Non-invasive. Both these mode contain different ventilation modes. Some of them are common in both.

The characteristics and parameter set of some of the modes are given below.

**2.5.1 Invasive**

The kind of ventilation is a frequent reason for admission to an intensive care unit. Invasive ventilation following endotracheal intubation (place a tube through the mouth to trachea) is used to assist-control, synchronized intermittent mandatory ventilation, and pressure support ventilation [20] [21]. The main three modes are described below.

**2.5.1.1 Volume Control (VC)**

The objective of the Volume Control mode is to deliver a preset Tidal Volume with a constant flow during the preset inspiratory time and at the preset Respiratory rate. Volume Control ventilation is primarily used when the patient has no spontaneous breathing at all. During the pause time both inspiratory and expiratory valves are closed and no flow is delivered to the patient.

The following parameters are set VC mode:

- Tidal Volume (ml) or Minute Volume (l/min)
- Respiratory Rate (b/min)
- PEEP (cmH2O)
• Oxygen concentration (%)
• I:E Ratio or Ti (Inspiratory time in second)
• Pause time (%/Second)
• Inspiratory rise time (% of breath cycle time / second)
• Trigger sensitivity

2.5.1.2 Pressure Control (PC)

In this controlled mode of ventilation the ventilator delivers a flow to maintain the preset pressure at a preset respiratory rate and during a preset inspiratory time. The pressure is constant during the inspiratory time avoiding unnecessarily high peak airway pressures and the flow is decelerating. If for any reason, pressure decreases during inspiration then the flow from the ventilator will immediately increase to maintain the set inspiratory pressure. The PC level above PEEP will result in the Peak Pressure. The controlled respiratory rate and I:E ratio is enabled in this mode.

The settable parameters in this mode are:

• PC (Pressure Control level) above PEEP (cmH2O)
• Respiratory Rate (b/min)
• PEEP (cmH2O)
• Oxygen Concentration (%)
• I:E Ratio or Ti (Inspiratory time in second)
• Inspiratory rise time (% of breath cycle time/s)
• Trigger Sensitivity

2.5.1.3 Pressure Regulated Volume Control (PRVC)

PRVC is a controlled mode of ventilation which combines the advantages of Volume Controlled and Pressure Controlled ventilation. The ventilator delivers the preset Tidal Volume with the lowest possible pressure. The first breath delivered to the patient is a Volume Controlled breath. The measured plateau pressure is used as the pressure level for the next breath. For the following breath, this pressure is constant during the set inspiratory tie and the flow is decelerating. The set Tidal Volume is achieved by automatic, breath-by-breath pressure regulation. The ventilator adjusts the inspiratory Pressure Control level to the lowest possible level to guarantee the preset Tidal Volume, in accordance with the mechanical properties of the airways/lung/thorax. It also delivers a controlled Respiratory Rate and I:E Ratio.

The settable parameters are:

• Tidal Volume (ml) or Minute Volume (l/min)
- Respiratory Rate (b/min)
- PEEP (cmH2O)
- Oxygen Concentration (%)
- I:E Ratio or T_i (Inspiratory time in second)
- Inspiratory rise time (% of breath cycle time/s)
- Trigger Sensitivity

### 2.5.2 Non-Invasive Ventilation (NIV)

NIV is to be used on non-intubated patients. It might be patients ventilated with a facemask as well as nasal prongs. There are mainly two differences compared to Invasive Ventilation:

- Low resistance since there is no tracheal tube.
- Leakage.

These issues have to be dealt with in NIV.

#### 2.5.2.1 Ventilation Modes in Non-Invasive Ventilation

The ventilation modes in NIV are:

1. NIV-PC, which is basically Pressure Control with leakage compensation.
2. NIV-PS, which is basically Pressure Support with leakage compensation.
3. NIV-NAVA, which is basically NAVA with leakage compensation. In NIV NAVA, the electric activity of the diaphragm is used to trigger and regulate on.
4. NCPAP. NCPAP is intended for premature infants breathing through Nasal Prongs. In NCPAP the ventilator delivers enough flow to maintain the set CPAP pressure. There is no patient triggering mechanism. NCPAP cannot be used in Adult.
3 Related Research, Real world implementation and MBT user survey

3.1 Related Research

Though the model-based testing is not an aged methodology, lots of research works were performed in this field both educationally and commercially. At the beginning phase of our literature review, we have gone through a number of papers, articles, journals, blogs, etc. Our motive was to get idea about how MBT is performed in different software industry and what is the prospect of it. The survey section gives an idea about the prospect of the model based testing. This section will give the summary of some papers that helped us to get theoretical idea.

In paper [22] author motivated how MBT is useful for automotive embedded systems and introduced TPT as a test tool for model based testing. It can be noted that though our target system is an embedded system, we used simulations of different parts of the actual system as our target. According to the author, graphical models and simulation of such models allows engineers to find a common functional understanding early in the design phase. This paper was a good start for getting theoretical idea about implementing MBT in a certain type of product and motivated to go through a tool.

In the planning phase of our thesis it was decided that a comprehensive GUI will be the first layer through which input will be given. In this paper [23] author demonstrated how to leverage model based testing practices in system testing through a GUI. The author proposed LTS with action words for testing whereas normally UML diagram, state machine is used for modeling.

In paper [24] Model Based Testing is used for Embedded Session and Transport Protocol to find bugs arising from concurrency. Author mentioned that testing of this type of system is very hard because beside exhaustive verification and proving for correctness, problems are augmented by the fact that it is not possible to force an executing parallel system to a certain state as timing and scheduling issues that cannot be influenced from outside of the implementation affect the behavior of the system under test.

In addition, there are some famous commercial and open source tools in the field of model based testing. The companies which have built the tools have done lots of their own research. As example,
three names can be mentioned: Conformiq, PikeTec, JSXM. In [25] [26] [27] downloadable documents with significant amount of information is available.

3.2 Real World Implementation

Model-Based testing has started its journey to the industrial level a little over 15 years ago [28]. Within a few years it made a strong position in mainstream software industry giants like IBM, Microsoft [12]. Model Based Testing has become popular in different kinds of software industries particularly telecommunication, embedded, automated systems or more specifically the systems that need to be used for very long time. In this section, some example situations are given:

3.2.1 Model-Based Testing at Microsoft

Microsoft is now at their third generation of model-based testing. At the first generation their Internet Explorer team built a MBT tool, called Test Model Toolkit (TMT). It uses simple finite state machine models, which are written as transition tables (tables of Start State, Action, End State triples and six test generation algorithms. The team won the testing best practice award inside Microsoft in 2001, and by 2004 was in use by at least 20 teams within Microsoft. Their experience with TMT was that it reduced the time to develop automated tests, made it easy to generate more tests, increased the code coverage, and uncovered many specification and implementation bugs. For example, the BizTalk team had spent 8 weeks writing a test suite manually, but in 1 week with TMT they generated a larger test suite that increased code coverage by 50 percent. The second and third generations of MBT tools is ASML/T7 and Spec Explorer. Spec Explorer is used daily by several Microsoft product teams. Spec Explorer discovered 10 times more errors than traditional manual testing, while the time to develop the Spec# model was about the same as the time taken to develop the manual test cases. The model-based testing also increased the coverage of the implementation code from 60 to 70 percent [12].

3.2.2 Model-Based Testing in Embedded System

In the case of software bugs in embedded in mobile devices like mobile phones, bank cards, passports, and health cards, it is not possible to patch the application without asking the owner to return the smart card. Therefore, product quality is a major issue and the level of testing is very high in this industry. Embedded systems like Smart card software can be tested in a simulated environment or tested on-card by using various hardware and software test harnesses to automate the test. Though the test automation problem is already solved in the smart card industry, but a huge effort is required for test design. By designing means finding the scenarios, writing different tests, their priorities and sequence designing. In addition, the general software development maturity level in the smart card industry is
high. This implies strong expertise in traceability issues, from requirements to code and test development. Model-based testing helps to industrialize this kind of process by automate the design of tests and connect to the existing test execution tools.

3.2.3 Model-Based Testing in Automotive Industry

Model-based technologies allow the development of high-level models that can be used for simulation in very early stages of the development process. This in turn is important since automotive development is an interdisciplinary business with software, electrical, and mechanical engineering aspects inextricably entwined. Graphical models and simulation of such models allows engineers to find a common functional understanding early in the design phase. So, model-based development improves communication within development teams, with customers, or between car manufacturers and suppliers [22].

3.3 MBT User Survey

This sections shows some survey reports about MBT from different view so that it is easy to understand the potentiality of model based testing over other testing methodologies.

An online survey was announced to the attendees of the first Model-based Testing User conference, held October 18-20, 2011 in Berlin. Calls for participation were also posted on MBTUC blog site and in public online forums: Linkedin.com Model-based Testing user group, Spec Explorer Forum, IBM Rational forum, the UseNet group comp.software.testing, and SQA Tester forum. The call was also emailed to several dozen participants of Advances in Model Based Testing (AMOST) workshops and others with an interest in MBT. The purpose of this survey is to learn about how model based testing is being used. One important question was: “Overall, how effective do you think MBT has been?” The graph below illustrates the responses from the attendees:

![Figure 3.1: Overall effectiveness of MBT](image)
Three out of four respondents see MBT as either moderately or extremely effective. No respondent rated MBT as ineffective, in any degree. In this paper [28] the results and analysis are given in very details from different point of view.

Another survey was performed by a student as a part of masters’ thesis at Linköping University for a very famous industrial MBT tool named Conformiq Qtronic. It was a comparison survey between manual testing and MBT to measure the efficiency for three kinds of SUT. They are:

- ATM Machine
- Calculator
- Inventory System

The below chart shows the comparison between efficiency of MBT and manual testing:

![Comparison efficiency between MBT and manual testing](image)

**Figure 3.2: Comparison efficiency between MBT and manual testing [29]**

The three SUTs is the chart are chosen from small system (Calculator) to semi-Large SUT (Inventory System). From the chart we can see that for every case the time to develop the testing environment with MBT is less than manual testing. It is also observed that we get larger number of test cases using MBT than manual testing. Details are given in paper [29]

In addition, in the paper [30], a survey on MBT was performed from different point of view. In fact, those surveys and real world examples motivate the fact that the potentiality of using MBT methodology in future software testing automation is pretty high. There is another tabular sheet generated by Software Acquisition Gold Practice [31] where Model-Based testing is evaluated by grade with respect to different characteristics.
4 Design Description

For implementing model based testing methodology in Maquet's existing test framework, an extension was necessary. Therefore, as a final output of our masters’ thesis we developed an application. This application is an extension to the existing framework that provides automation facility to that framework. In summary, it can be said that it is the complement of the existing framework that provides the possibility to upgrade the existing unit test based framework to automated test framework [32]. This chapter gives a design description of our application.

4.1 Application Overview

MBT application is a testing application for the patient ventilator. It has been designed based on Model-based testing paradigm: testing a large, detailed and complicated system based on a small, agile and accurate model of that system. This application partially simulates the behavior of a patient ventilator system in the case of dependencies between parameters. This way, it generates expected results and compares with patient ventilator results to find differences and potential faults.

4.2 Application Architecture

This application can be considered as a three layers application. The first layer is GUI, the second layer is the engine of the application which is responsible for generating test cases, and in the third layer the adapter is placed. The engine layer consists of combinatorics, model and application's logic.

![Figure 4.1: Application design in layer](image)
The adapter layer acts as a connector to the existing test tools. Figure 4.1 illustrates these layers.

In the GUI level, this application utilizes a user interface resembling the patient ventilator user interface. It enables to select different ventilation modes and then their corresponding parameters. For each parameter, the user can select a range of values to be tested. Then the range of parameters is sent to the combinatorics to generate unique combinations of parameter's values. In the next step, this combinations feeds to the model to their validity be verified. The model checks each combination of parameter's values and determines whether this combination is valid (settable on the ventilator) or not. Using the adapter, the results of model will be run on the target system (system under test which could be a ventilator simulator or a ventilator machine. Currently the target system is the patient simulator). Next the result of running on target system can be compared with the model's output for finding differences.

4.3 Application IO

The data is transferred in each layers of the application to the next layer through XML files. GUI gets the user input values consists of a ventilation mode with all parameters and its values as well as configuration and settings (such as patient category, etc.). Then saves all this data in a XML file as a test set. Having arbitrary number of test cases XML files which is called a test suit, the application sends this test suit to the engine. Next, the engine generates results for test cases and saves them into separated corresponding XML files. The adapter reads this XML result files and run them on the target system.

The application correlates each test case XML file with corresponding results XML file from both model and target system using unique ID. The unique ID is a part of each file name; moreover, it is written on the content of XML files.

4.4 GUI

In addition to enabling to select different ventilation modes and the range of parameters, it provides to select between many different configurations and settings. GUI also provides a way to save and load all test set and result XML files.

4.5 Application Engine

The second layer of this MBT application is the engine layer. It has two sub layers. They are described below:
4.5.1 Combinatorics

The combinatorics is responsible to generate unique combinations of parameter's values. In order to limit the number of combinations to an arbitrary maximum value, it assigns a weight to each parameter according to its range of values. It uses these weights to generate proportional number of values for each parameter. Next, for these proportional number of values the combinations will be created. Each combination contains a list of parameter's ID and its corresponding value pairs. Every single combination is a test case. Therefore, output of combinatorics is a list of test cases.

4.5.2 Model

Model is responsible for modeling and simulating the behavior of ventilator machine. It receives the list of test cases (generated by combinatorics) along with a set of critical configuration data such as patient category, time unit (percent or second), volume unit (tidal or minute) and some more. For each test case it checks each parameter's value to verify if the value is valid and settable or not. The output of the model is a result XML file containing all test cases and validity results for each test case.

4.6 Adapter

The adapter responsibility is running the result of model on the target system. Therefore, for every resulting XML file generated by the model, it extracts test cases and after running on the target, writes the results on another XML file which is called target or AVA XML file. This file has the same format as model XML file.

4.7 Underlying systems

The adapter communicates with the target system which currently is a simulator (called Tellus Patient Simulator) for sending and receiving result. But user cannot directly interact with it because it has no direct interface. An internal test framework named AVA (Advanced Ventilation) works as an API to serve this purpose on behalf of the end user. It is a python based API that run in Linux environment. Test scripts are written maintaining a particular format. There are different methods and variables in AVA framework. By passing the parameter through the API with user can send data to the lower layer and receive the status. The user can use those data to reach in a decision which in our case the model does. In addition, to provide more facilitation another high level framework is developed named HEVT (High End Ventilation Tool) where a bunch of methods are put together to serve a high level purpose, for example, changing the ventilation mode or patient category. So the user do not need to take care of the lower level details in AVA. But HEVT is not a part of AVA. For formatting messages
in application layer, Protocol Buffer developed by Google is used [33]. Protocol Buffers allow you to define simple data structures in a special definition language, then compile them to produce classes to represent those structures in the language of your choice [34]. In this framework protocol buffer for python is used. Overall, TCP/IP protocol is responsible for communication between AVA and binary files of patient simulator. More details are given in implantation chapter.

4.8 Choosing Tool

After designing the system through layered approach, the next step was to choose the tools. This preformed in two steps. First is choosing the MBT tool and then to choose the platforms.

4.8.1 Choosing MBT Tool

Choosing of most suitable MBT tool for our system is one of the most important tasks. If the tool is chosen without proper research on both target system (Patient ventilator) and different MBT tools and the mistake is realized after the half of the implementation is done then we have to backtrack again to implement all those steps with other tool. Our previous experiences in programming field are also taken into consideration for this purpose. The details are given below:

4.8.2 Requirement:

Our system falls in data based criteria (rather than transition-based) that are testing of one-value, all-values, boundary-values, and pair-wise values, refer to the selection of input values when creating concrete test cases from abstract test cases [35]. After going through a taxonomy of MBT according to [14], we categorized our target system requirement into following:

1. **Scope**: The scope of the model will be input-output based. We have to know the input-output behavior of the system

2. **Characteristics**: The characteristics of the model are Deterministic and Discrete. Details are following:
   a. All the states are known and it can be defined.
   b. Timing issue will not be considered strictly
   c. The model will be focused on event-discrete system, real time scenarios will not be considered

**Model Paradigm**: Our system consist few states. But a transition for going one state to another state we need many input data. We will choose Data-Flow Notations because the target system’s concentration is on the data flow rather than the control flow. It can be noted that a very important
quality of using transition-based notation is that it is more descriptive and comprehensive from viewer point.

**Test Selection criteria:** The requirement of our project is to choose few test values from a large data space. We need to group our data to equivalent classes. Value from each will work as a representative for every group. So we will use Data Coverage Criteria. We will also use Fault-Based criteria because they are mostly applicable to SUT models. Our goal of testing is to find faults in the SUT.

3. **Technology:** We will use Model-checking technology for verifying or falsifying properties of a system. The general idea of test case generation with model checkers is to first formulate test case specifications.

4. **Online/Offline:** We need a tool which supports offline test generation

In addition, two other requirement is important for our target system which is:

1. Tool must have strong data type and dynamism in variables.
2. Preferable language support should be Java, python, XML (according to project requirement)

There are some good commercial and open source tools for MBT. Our intention for going through these tool to get a practical idea of MBT steps and then choosing the best tool for our target system. A small description about some of them are given below:

### 4.8.3 Commercial Tools

For most of the commercials tools there are GUI tools for modeling the system (FSM or EFSM). In that modeling they use graphical symbols for draw states, transitions etc. Then their proprietary language is used to explain the pre-conditions, post-conditions, variable declarations etc. Some of the examples of these languages are: QML, mGraph etc. Then set of test cases can be found by test case designer by defining testing goals such as requirements, states and transition coverage. Below the properties of some commercial tools are described:

**AETG:** This tool is good for combinatorial testing to reduce the large number of possible combinations of input variables to a few ‘representative’ ones. We got allpairs, SmartTest, AETG. These are not recommended because of input only and that is why do not provide expected output. But allpairs can be used with customized conditions as combinatorics to select the pairs of input.

**JUMBL, MaTeLo:** These are also input data only tools and do not provide the expected response of the system. Test inputs are generated and SUT behavior is not modeled. JUMBL models are written in a transition-based notation for describing Markov chain usage models. A Markov chain usage model has a unique start state, a unique final state, a set of intermediate usage states, and transition arcs
between states. The transition arcs are labeled by the corresponding event and the probability of occurrence. Transition probabilities are based on expected use of the SUT. The JUMBL primarily uses an offline approach.

**SpecExplorer, Conformiq, CertifyIT (SmartTesting):** These tools have both RCP (Rich Client Perform) or they have plug-ins for another IDE. For example, SpecExplorer has plug-in for Visual Studio and Conformiq has plug-in for Eclipse. Some important properties of this group of tools are given below:

- SUT input–output model, which is typically composed from several simpler models
- provides the oracle for each generated test case
- Models are untimed and discrete. Non-determinism is supported by distinguishing controllable actions from observable actions—the latter may be generated spontaneously by the SUT.
- Multiple models written in these notations are composed to obtain the final SUT model
- Including data coverage of parameter values and the state space and structural model criteria such as covering all transitions. A regular expression can also be used as an explicit test case specification
- Uses bounded model checking and supports online and offline

**Test Optimal:** TestOptimal is a web based client server tool that tests desktop and multitier enterprise applications. A TestOptimal model is an FSM, created interactively while analyzing the web site being tested. It can also be imported in GraphML9, XMI10 and GraphXML11 formats. TestOptimal provides model validation, simulation and debugging support. It provides an XML based scripting language called mScript to connect (adapter/driver) the model to the SUT. A tester can test do scenario testing using mCase. TestOptimal provides multiple algorithms to generate test cases and supports online and offline testing. It can be used for stress, load and regression testing. TestOptimal automatically generates test adapter class skeleton where a tester can add function logic to run generated test cases.

### 4.8.4 Open-source Tools

**GraphWalker (mbt):** It is a tool for generating online or offline testing using FSM or EFSM. The FSM or EFSM can be made using independent tool. GraphWalker has syntax available in the site documentation for generating model with other tool and giving pre-post condition there. We used yED to make the FSM. The output is .graphml file which can be fed to GraphWalker along with a template (.template file). There is a command for that parsing and a .jar file should be available in that current directory. That .jar file is also available in that site. The output is a stub written in java contains bunch of empty methods. These methods are needed to modify with declaring variables and expressions and from that stub test cases are generated [36].
**JSXM:** In JSXM, the difference between GraphWalker is that one can write the implementation of stub (called in GraphWalker) in that XML file which as a whole is known as the model of JSXM. The advantage is that it is more standard so it is more helpful for parsing. In addition, one does not need to worry about synchronizing the variable of model with variable of generated test cases variables because it is only written once in the model. JSXM transforms the tests to executable tests in the language of the implementation (e.g. JUnit) [37].

After studying requirement, taxonomy and tools, two ways are found:

One way can be drawing the model with a GUI Based tool and the source of the file must be a standard output for parsing by another plug-in, jar file or compatible software. This file can be parsing through a parser to generate stub. This stub can be some abstract methods or methods with only signature. In those methods variable or expressions can be declared. Here the advantage is that it will be more comprehensible to the target audiences.

Another way can be writing everything in the model using standard tags. Here from the beginning the states and transitions are defined using tags. It is possible to define methods, variables (pre-conditions, post-conditions) etc. inside that file. When the outputs are generated then corresponding methods are generated for variables and parameters and the range of values can be given there. The advantage of this method is one does not need to write variables two times (first in model and another in stub) and does not need to think about synchronizing those variables (model and stub).

After going through all these we decided not to use those traditional tool. The reason is we do not know whether they will support our requirements or not all the way through the project. Besides they support less dynamism and flexibility, particularly in the context of variable type. As our tool will be only used in a particular framework and tool will not be used commercially, we preferred to build a new tool by our self which will be most suitable for organization’s internal test framework.

### 4.8.5 Choosing platform

This step was comparably easy because choosing the language was written in masters’ thesis requirements, which are java and python. For modeling, we decided to use java because most persons in this related background are familiarized with it. It has a strong and dynamic data structure and is the most famous object oriented language. Different solutions are easily available online for this platform.

The organization’s internal test framework was in made in python. Test scripts are also written in python following unit test framework. That is why python is chosen for making the adapter. The test framework runs only in Linux environment. That is why Linux shell scripting is chosen for integration purpose.
5 Implementation

MBT application imitates the patient ventilator in many ways. Consequently it includes a number of concepts exist in the ventilator. These concepts such as Ventilation modes, Ventilation types, Parameters, Patient categories are implemented in the application. In this chapter, by user we mean the users who will use the MBT tool.

5.1 Overview

The following activity diagram briefly illustrates the whole work flow of the MBT application’s implementation step.

![Figure 5.1: Iteration of MBT Application through Test suits](image)

From the figure we can see that the activity diagram starts with a set of test suits or configurations as its input. Those inputs are XML files produced by the GUI of this application. An XML file has all the configuration and patient category information which the model use to configure the patient simulator. After that every test cases are checked by the model one by one until the test cases are finished. The action state “Check test case” is illustrated in the following activity diagram.
Figure 5.2: Detailed view of Check test case action state of figure

From the figure, we see that at the first state it just check the parameter and write the result in XML file. This is the output of model. Parameters and their values from test cases are then sent to target system. In this point a validity check is performed. If the parameter is valid then “True” is returned. In that case, model sets the parameter and its corresponding value. The model again performs the same steps for the rest of the parameters in that test case. Meanwhile, if the validity check returns fail then another check is performed. This check is performed to see that whether the set of parameters in the test case is checked by the model for first time or it has done before. Every test case contains in XML file has a tag that determines whether the test case is unique or reordered. By checking the value of the tag model can recognize the set of parameter. If one parameter in a test case is invalid then the model set the value of the tag to reordered. Then the model change the setting precedence of the parameter set and checks again. If still at least one parameter’s value is found invalid, then model discards the test case and proceed with another one until all the test cases in the test suit is finished.

Here, it is worth mentioning that the validity check for parameter is the main engine of the model. The model implements equations for all the parameters and their dependencies with other parameters. Those complex equations are described in the design documentation of the patient ventilator. This information is private property of the organization. That is why those are not described in this documentation. In addition, the combinatorics is used to generate test cases with all unique combinations for a set of parameters each with a different range. The code implementation is described in the following section.
5.2 Coding implementation

The implemented packages and classes will be presented in the following.

5.2.1 Domain package

This package contains the following classes.

- ** Enums**: Defines all required enumerations (a data type) in the application.
- ** Category**: Represents a patient category (Adult, Infant or Neonate) in the application.
- ** Parameter**: Represents a ventilation parameter in the application.
- ** Mode**: Represents a ventilation mode in the application, each Mode instance contains a list of Parameter instances.
- ** VentilationType**: Represents a ventilation type in the application, each VentilationType instance contains a list of Mode instances.
- ** Configuration**: Encapsulates all configuration settings for the application.

5.2.1.1 Parameter

The list of the most important properties of this class are:

- ** ID**: parameter's ID in existing system and documents, e.g. "PRESS_ABOVE_PEEP" for pressure above PEEP
- ** step**: the steps between parameter values, for instance in PRESS_ABOVE_PEEP step is 1, it increases or decreases one by one
- ** minValues**: an array of minimum values for parameter in different patient categories. The first element contains minimum value in Adult mode, the second element contains minimum value in Infant (Pediatric) mode, and the third element contains minimum value in Neonatal mode.
- ** maxValues**: an array of maximum values for parameter in different patient categories. (Similar to minValues)
- ** minValue**: keeps selected minimum value for parameter (to be used in Combinatorics)
- ** maxValue**: keeps selected maximum value for parameter (to be used in Combinatorics)

5.2.2 GUI

The application GUI has been implemented using JavaFX [38]. MBT application uses Cascading Style Sheets (CSS) to customize look and feel of the application [39]. It also utilizes JavaFX animations to make the application more interactive and attractive.

There are three packages containing classes of GUI:
- **ui package**: the main package consists of all classes for creating and handling application screens and windows. This package also includes the Ui class which is starting point of application.

- **ui.components package**: graphical components which are used in the application as a base class or implements a common graphical component.

- **ui.doubleSlider package**: this package and all its classes has been added to the application to provide a control component for selecting a range of values. Due to the lack of such components in JavaFX control collection, this package has been downloaded.

### 5.2.2.1 ui package

This package contains the following classes:

**Ui class**: In a JavaFX application there is a start method which is the entry point for running the application. This class overrides start method for the application. It is also responsible for initializing and resetting the whole application resources and configurations.

**ScreenPreLoader class**: Displays the application pre loader.

**ScreenMain class**: The main screen of application. All windows will be add to this screen.

**Windowing classes**: There are seven different classes as windows of application each one responsible for displaying of the window and handling the logic of the application. The constructor of each class is responsible for adding controls. There is also a setHandlers method in each class which set handler methods for controls. This way it has been tried to separate the GUI code from the logic code. All instances of these classes will be added to the main screen.

**ScreenRun**: Displays run screen, is the entry point for starting the application engine. All the methods needed for starting combinatoric, model and adapter are defined and implemented in this class. This class is takes one test set XML file and sends the data to combinatoric and then to the model and then to the adapter for running. Then the same process will be done for the rest of test set XML files one by one.

**Animations class**: Provides animations for the application. It contains static methods which implement animations for a control.

**AppGraphics class**: This class defines all sizes and graphical icons and images in the application. In this way when a size or graphical components changes the only effected class will be this class. Therefore, every change in this class will be reflected on the whole GUI.
5.2.2.2  ui.components package

There are some controls and widgets needed by the application which still are not a part of JavaFX control collection, for instance a dialogue window. This kind of graphical widgets are defined in this package. This package introduces the following classes:

Window class: This class is the base class for all Windowing classes in the ui package.

Dialogue class: Implements a dialogue control for the application.

ProgressCircle: Implements a progress circle, it is a kind of progress bar control.

ButtonsGraphic: This class implements a graphical control for many button controls of the GUI. It introduces many constructors for customizing a button control.

5.2.3  GUI resources

The resources including CSS files, fonts, images and icons are bundled in the skin package. Two font files are used in the application which enables to use “Century Gothic” font. These font files have been downloaded from Google font collection [40]. There are two different CSS files in this application:

defaults.css: this file creates a completely different and modern look and feel for the application. This styling file introduces or modifies around 200 different style classes to reach ideal style for the GUI.

original.css: this styling file tries to design look and feel similar to the Tellus patient ventilator GUI. It also introduces or modifies around 200 different style classes.

5.2.4  Application data and state

The state package contains two classes. AppData class in which all applications data and settings are defined and maintained. In addition, AppState class holds the state of application settings and configurations.

AppData class: MBT application's data such as all implemented ventilation modes, all implemented parameters, patient categories and so on are defined and instantiated in this class. Moreover, the constructor of this class enables to adjust different parameters corresponding to different time and volume units’ configuration.

AppState class: This class is responsible for keeping the applications’ states. It utilizes JavaFX binding facilities to propagate changes in state class to all other bound objects. If any objects changes a setting in the application it must changes this settings in the AppState class. This changing will be automatic using bindings or with changing the setting on the AppState directly. In this way this class
always keeps the last updates and changes to the application. Some of this settings includes workspace
directory, time and volume settings, number of test sets and so on.

5.2.5 Combinatoric

This class implements the combinatorics for the application. It extends Task class to be able to run in a
different thread in the case of large number of combinations and calculations. The other advantage of
extending Task is to update the progress of generating combinations.

The Combinatoric class generates unique combinations from a list of input Parameters. There is an
inner CombinatoricParameter Class which is used to add more fields to Parameter and facilitates
calculations. This class also receives a maximum number for generated combinations. In the first step,
number of all possible combinations is calculated for given parameters. If the number of all possible
combinations is equal or less than maximum number then all combinations will be generated. Else,
since the number of generated values for each parameter is restricted, for every input parameter only
specific number of values are selected for generating combinations. The combinatoric assigns a weight
to each parameter according to its size (all possible number of values). Then calculates required
number of values for each parameter which are used for generating combinations. The generated
values for each parameter and finally, it generates combinations with list of values for each parameter.

In other words, there are three steps for generating combinations: first finding how many values should
be taken from each parameter's range of values, second selecting these values, and third generating
combination using these values.

5.2.5.1 Number of values for each parameter

Considering the following definitions:

- N: number of parameters
- MAX: maximum number of combinations
- Min (i): minimum number for (i) parameter (lower value of parameter's range)
- Max (i): maximum number for (i) parameter (upper value of parameter's range)
- Step (i): step size for (i) parameter (block increment)
- sizeOf (i): number of all possible values in a (i) parameter's range
- maxSize: the parameter with the biggest size (i)
- w (i): assigned weight to (i) parameter
- sizeToWeight (i): required number of values of (i) parameter which are used for generating
  combinations
$$sizeOf(i) = ((max(i) - min(i)) \div step(i)) + 1$$

We need a factor number that:

Since, $w(i) \leq 1 \Rightarrow \prod_{i=1}^{N} w(i) \leq 1$

So, $\rightarrow MAX = \left( \prod_{i=1}^{N} w(i) \right) \times factor^N$

In above formula the only unknown value is factor, to count that:

$$factor^N = \frac{MAX}{\left( \prod_{i=1}^{N} w(i) \right)} \rightarrow \log_{factor} \left( \frac{MAX}{\prod_{i=1}^{N} w(i)} \right) = N$$

Since,

$$\ln \left( \frac{MAX}{\prod_{i=1}^{N} w(i)} \right) \rightarrow \ln(factor) = N \rightarrow \ln(factor) = \frac{\ln(MAX)}{N}$$

And because inverse function of natural algorithm is exponential function:

So for each parameter:

$$sizeToWeight(i) = w(i) \times factor$$
All parameters must have at least one value in each combinations so required number of values of (i)  
parameter (sizeToWeight (i)) must be at least one. Therefore if sizeToWeight (i) is less than 1 (one)  
the following will be done:

1. The sizeToWeight for this parameter will set to one (1)
2. This parameter will be removed from the list of parameters
3. The process of calculating factor and then sizeToWeight(i) is repeated again for new list of  
parameters
4. This process is repeated until all sizeToWeight(i) are more than one (1)

5.2.5.2 Generating values for each parameter

In this step and with a given required number of values (sizeToWeight (i)) for each parameter these  
values are generated. Each parameter has a field which determines whether generated values should be  
random values or not. In this case required number of values for the parameter are generated randomly  
between min (i) and max (i).

In the case on non-random values, the combinatoric selects the min (i) value as the first generated  
value and generates the rest of values by increasing a fragment number. The fragment is counted as  
following:

\[ \text{fragment} = \left( \text{sizeof}(i) \div \text{sizeToWeight}(i) \right) \times \text{step}(i) \]

In this way it makes sure that selected values are from the whole range of parameter's values. Then this  
generated values for each parameter are added to a list. The list is used to generate output  
combinations.

5.2.6 Model

Similar to the Combinatoric, this class also extends Task class for the same reasons. This class  
implements model for the application so it contains all required data and methods. The equations and  
variables’ values that model implements are private property of the organization. That is why those are  
not described in this document.

5.2.6.1 Model data

This class contains all implemented parameters in the MBT application along with default values for  
each parameter. In other words, for every parameter implemented in AppData class there is a similar  
parameter in model class. These parameters are correlated through their Ids which is also the same as  
parameters' Ids in existing systems. Inside models class there is an inner class VentilatorParameter
which is a customized class for representing a Parameter in Ventilator class. The VentilatorParameter contains the following fields:

**ID**: Parameter's ID which is corresponding to the ID of parameter in the whole MBT application as well as existing tools and devices like AVA.

**value**: The value of parameter.

**defaults**: An array of three values which keeps default values for each patient category. (index0: ADULT, index1: INFANT, index2: NEONATAL).

The default values are extracted from the Tellus Patient Simulator. They must be updated with the target system to get correct results from model.

This class also receives an instance of Configuration class during initialization. Using the information provided by Configuration instance, the model will reset default values and will do calculations and resolve dependencies between parameters.

**5.2.6.2 Model Behavior**

The model receives a list of test cases (combinations) to be verified. For each test case it first resets default values. There is a method called resetDefaultValues(). This method resets the default values for each patient category. It also sets different default values for parameters whenever is needed by a specific ventilation mode.

Then it checks parameters in the test case one by one. For every set-able parameters there is a corresponding check method which is called to validate the parameter's value. Each test case contains a number of pairs of (parameter’s ID, parameter's value) which represents all parameters available in a specific ventilation mode. For validating a test case first all parameter's values are reset to default values, then a check method is run to validate each parameter's value. If parameter's value is valid then its corresponding field will set to this valid value and the checker method returns true and other parameters will be checked. Else if parameter's value is not valid the checker method returns false and other parameters of this test case will not be checked. In the case of invalid parameter's value, there is a second attempt to validate the parameter's value. Doing this for the current test case, the invalid parameter will be moved to the end of list of parameters (reordered) and the test case is validated with this new order again.

All this actions are written in the model XML file which is passed to the target device for running.
5.2.7  Application IO system

When MBT application is installed on a computer a workspace is created. When MBT application starts to run a unique workspace is created inside the workspace for writing and reading XML files. The name of workspace and unique workspace is maintained in the AppState class.

5.2.8  Workspace structure

Inside the unique workspace the following directories are created:

Input: this directory is used for reading and writing test set XML files. In fact every selected test set is saved in this directory and then during running these files are loaded and data is sent to the combinatoric. When a test set is removed from the application, the corresponding XML file will be deleted from this directory.

ModelOutput: this directory keeps the results XML file generated by the model. All contents of this directory will be removed for a new run of test cases.

AvaOutput: this directory keeps the results XML file generated by the target device (which currently is Tellus Patient Simulator). All contents of this directory will be removed for a new run of test sets.

Comparison: it keeps the comparison spreadsheet file generated by the adapter. All contents of this directory will be removed for a new run of test sets.

Logs: it keeps log results generated by the model. These file are mostly used for troubleshooting the model.

MBT application assigns a unique ID to each test set. This unique ID is inserted as a part of name of each file and in a tag inside the XML file content. It is used to correlate each file existing in the above directories with each other. After a successful MBT application for a number of test sets, all folders contain the same number of related files. When MBT application is closed this folders inside the unique directory would be left unchanged. Then in the next running of the application the unique directory and all directories will be created again.

5.2.9  Schemas, validation

For implementing XML IO system, two XML schemas has been designed. These are placed in the schemas package:

Input.xsd: this file is a schema file for a test case XML file. The generated XML file according to this schema keeps all information about a specific ventilation mode as well as some configuration data.
**Testcases.xsd**: this file represents a schema for generated results from both model and target device. These two XML files have the same structure to be comparable with each other.

The application uses these schemas to validate a XML file when it is loaded in the application. So before any operation on XML files first they will be validate, if they are valid then extracting information from them is started.

### 5.2.10 XML handlers

The xml package contains three classes which facilitates XML files operation in the application. These classes provides utility methods for validating XML files, writing and saving XML files and parsing and extracting information from XML files; moreover, they provide methods for comparing XML files.

**XmlInputHandler**: This class handles test case XML files. The InputXmlNode inner class supports this class to write and read XML tags properly.

**XmlResultHandler**: This class handles result XML files. The ResultXmlNode inner class supports this class to write and read XML tags properly.

**Testcases**: because of complexity of results XML files, this class implements an object equivalent to the result XML file and provides required behaviour for comparing these XML files.

### 5.2.10.1 XML files samples

The figure below represents a sample test suite XML file. There are following tags and attributes in this file:
Figure 5.3: Test suite XML sample

**input**: This is the root tag of the XML file. Inside this tag there are two attributes.

**xmlns:in**: This is the namespace of the XML file. It validates the XML file if necessary.

**id**: This tag is used for tracking different XML files related to the same session. The figure above shows the sample XML file of test suite selection. Two other XML files are generated by the application for a particular run. They are for model output and AVA output. The value of the id attribute is the same in those other XML files which help the application to identify XML files related with a particular session.

**time-unit**: This is a tag for start-up configuration parameter. The value PERCENT means that time will be calculated with the percentage unit. For example, for inspiration time the parameter will be
INSP_RISE_TIME_PERC. The other option is SECOND. In that case, the parameter will be INSP_RISE_TIME_SEC.

**volume-unit:** This is also a start-up configuration setting tag. The parameter will be TIDAL_VOL for the value TIDAL. The other option is MINUTE. In that case, the parameter will MINUTE_VOL.

**run-duration:** This tag defines the amount of time up to which a particular test case will be run in running mode of the ventilator. The unit of time is minute.

**ventilation-type:** It is an initialization parameter setting tag. It defines what will be the ventilation type for every test case in this test suit. OFF means the type will be invasive and ON means the type will be non-invasive.

**mode:** It is the ventilation mode for this whole test suit. This initialization tag defines the ventilation mode for the whole test suit.

**category:** This initialization parameter selects the patient category of every set of test case in the test suit. The three options are: ADULT, INFANT and NEONATE.

**maxNumberOfTestCases:** This tag is for a utility option of MBT application. After selecting the parameters' value (or a range of value) the user can limit maximum number of test cases with the combination of values.

**parameters:** This tag contains all the parameter tag inside it.

**parameter:** For test suit XML file it has one attribute and three other tag inside it. The attribute 'id' contains the ID of the parameter. The first two tag inside it namely min, max contains the lower input value, the higher input value of a range. The third tag inside (random) determines where random values will be selected or not from the range.
Figure 5.4: A portion of XML Output by Adapter

```xml
<res:testcases xmlns:res='urn:testcases' id='3e95dbd1-56ab-4fdb-86f7-efc06bd04b2d' type='AVA'>
  <time-unit>PERCENT</time-unit>
  <volume-unit>TIDAL</volume-unit>
  <run-duration>9</run-duration>
  <ventilation-type>OFF</ventilation-type>
  <mode>PRESSURE_CONTROL</mode>
  <category>ADULT</category>
  <testcase id='1' type='UNIQUE'>
    <parameter id='PEEP' value='1.0'>true</parameter>
    <parameter id='CMV_FREQ' value='4.0'>true</parameter>
    <parameter id='PRESS_ABOVE_PEEP' value='120.0'>false</parameter>
  </testcase>
  <testcase id='2' type='REORDERED'>
    <parameter id='PEEP' value='1.0'>true</parameter>
    <parameter id='CMV_FREQ' value='4.0'>true</parameter>
    <parameter id='I_E_RATIO' value='4.0'>true</parameter>
    <parameter id='INSPIR_TIME_PERC' value='20.0'>true</parameter>
    <parameter id='PRESS_ABOVE_PEEP' value='120.0'>false</parameter>
  </testcase>
</res:testcases>
```

Figure 5.5: A portion of XML Output by Model

```xml
<res:testcases xmlns:res='urn:testcases' id='712b4d92-b12c-46d9-bd9a-6339e753807' type='MODEL'>
  <time-unit>PERCENT</time-unit>
  <volume-unit>TIDAL</volume-unit>
  <run-duration>9</run-duration>
  <ventilation-type>OFF</ventilation-type>
  <mode>PRESSURE_CONTROL</mode>
  <category>ADULT</category>
  <testcase id='1' type='UNIQUE'>
    <parameter id='PEEP' value='1.0'>true</parameter>
    <parameter id='CMV_FREQ' value='4.0'>true</parameter>
    <parameter id='PRESS_ABOVE_PEEP' value='6.0'>true</parameter>
    <parameter id='I_E_RATIO' value='0.1'>true</parameter>
    <parameter id='INSPIR_TIME_PERC' value='10.0'>false</parameter>
  </testcase>
  <testcase id='2' type='REORDERED'>
    <parameter id='PEEP' value='1.0'>true</parameter>
    <parameter id='CMV_FREQ' value='4.0'>true</parameter>
    <parameter id='PRESS_ABOVE_PEEP' value='6.0'>true</parameter>
    <parameter id='I_E_RATIO' value='0.1'>true</parameter>
    <parameter id='INSPIR_TIME_PERC' value='10.0'>false</parameter>
  </testcase>
</res:testcases>
```

The above two figures show the other XML files generated by the MBT application. Most of the tags are same like the test suit XML file and described earlier. There are few new tags and attributes:

type: This attribute is inside the root tag of the XML file named ‘testcases’. It identifies whether the XML file is generated from model or AVA.

testcase: The tag testcase shown in the file has two attributes. First one contains the id of the test case. The second describes whether the combination of the parameter is unique or the parameters are re-ordered.
parameter: In these XML files parameter tag is a bit different. Instead of containing the limit values it contains the id and one value of a parameter. The value field of the tag contains true or false which says whether the value is possible to set in or not in either the target system or model.

5.2.11 Adapter

The adapter is a set of Python scripts which enables the application to communicate with the target system in addition to some a java class which starts running those scripts. In the adapter package there is a TargetRunner class. This class extends Task class for the same reason as combinatoric and model classes. TargetRunner class prepares the required scripts for running on target system and creates the child process for running those scripts.

5.2.11.1 Adapter scripts

The scripts and their missions are listed below:

**runner**: This is a shell script which initializes underlying system for running test cases, after that it runs the MBT_adapter.py script to run test cases.

**killer**: This shell script finds all the processes that are running in the system by MBT application and kills those if the user wants to kill the application instantly while it is running test cases.

**MBT_io.py**: Inside this script three file paths are inserted, model result XML file, target (AVA) result XML file and compare XLS file. This scripts is rewritten each time for running a new test set.

**MBT_adapter.py**: This script contains the adapter code, it invokes MBT_io.py for getting the path of XML files. It takes the path of model results XML file, extracts test cases and run them on the target device. Meanwhile it creates the target result XML file and compare XLS file with the given path and names. It also uses GBP_converter.py script for naming conversion.

**GBP_converter.py**: It contains the required scripts to converting names compatible with underlying system.

All the above scripts resides on the script package in the code. But when MBT application starts those are copied to the AVA directory. AVA directory is the directory where the required AVA scripts and API are available and installed. This directory may change from a computer to another so there is a variable inside AppState class which keeps the default path of AVA directory. This path can be changed through GUI for different machines. The only script which is copied at the beginning of running each test set is MBT_io.py which keeps the XML file paths for that specific test set.
5.2.12 Adapter in detail

A test script is written with python code to run test cases in AVA. This test script is not a part of AVA. This test script follows python UnitTest framework. It contains a setup method which runs before every test method and a tear down method that runs after every test method. This methods are responsible for starting and stopping different processes in patient simulator through AVA. All required tests are performed inside the test method.

Adapter is basically a test script following unit-test format. For making adapter the structure of the AVA test script is followed. The difference between the usual test script and adapter is that the adapter extracts data from XML file and writes output to XML file instead of manually written parameters and their values in the python methods inside the script. Steps that are performed in the adapter AVA are illustrated below:

![Diagram](image)

Figure 5.6: Workflow of Adapter for a single test case
The steps are described below in details:

1. Setup value in configuration mode. All parameter and values are extracted from the output XML generated by model.
2. Go to standby mode.
3. Set the initialization parameter as given in the test suit (in this case XML file)
4. Set the parameters' ID and value of that particular test case in standby mode
5. Activate parameters' value in standby mode
6. Go to run mode
7. Keep the mode running for the given amount of time in test suit
8. Write output in XML and Spreadsheet file
9. Go to standby mode
10. Resetting the patient category and ventilation mode
11. Repeat steps 3 to 10 (if there are more test cases)

After executing all the test cases in the test suit, the GUI gives a notification of completion of execution. Then the user can access those raw files through GUI. The GUI shows the summary of the outputs by extracting data from the XML files and the spreadsheets

**5.2.13 Adapter utilities**

The Adapter also used some python libraries to generate excel file as output. xlwt library is used for generating excel output. It is a library for generating spreadsheet files that are compatible with Excel 97/2000/XP/2003, OpenOffice.org Calc, and Gnumeric. It has full support for Unicode. Excel spreadsheets can be generated on any platform without needing Excel or a COM server. xlwt-0.7.5 package version is used for adapter application. The only requirement is Python 2.3 to 2.7 [41].

The output spreadsheet file contains two pages. First page contains the summary report of the output. Basically this page shows the value in last column with RED font if there is any mismatch between model and AVA. This feedback can be used to determine the error in either model or AVA. The second page contains the detailed output of the parameters value before and after setting the parameter in both PENDING and CURRENT state in simulator. Last column in this page with RED font indicates that invalid values are prompt to set in the simulator.
5.2.14 Underlying systems

MBT application adds some layers to the existing test framework of Maquet named AVA. The underlying systems that the application uses to run are described below:

High End Ventilation Tools (HEVT)
HEVT is a framework which uses AVA below as an API. HEVT contains a bundle of classes and methods. This framework is built with the most frequent operations that are performed in AVA API to deal with patient simulator. There are two python file for that framework. They are core_tools.py and panel_tools.py. There is a class named Macro in core_tools.py file. Group of methods in this class is used to perform common system commands. Methods from this class is used for python adapter. It is worth mentioning that HEVT is not a part of AVA.

Advanced Ventilation (AVA)
AVA is a test framework that simplifies and unifies the procedure of writing advanced system and subsystem tests. AVA is written in Python and runs primarily on Linux systems, mainly because the original test targets are built as Linux ELF binaries [42]. AVA uses Google’s Protocol buffer in its application layer of the TCP/IP protocol stack to communicate with Patient Simulator. Porting to other platforms should be relatively simple, if the platform supports Python. To be able to use AVA, the host system needs to have Python 2.6 or 2.7 installed. AVA uses the Python UnitTest framework for its test cases. The UnitTest framework gives deep control of the test flow, and provides functions for separating test methods, setup code before each test method, and tear down code after each test method, pre-test setup code, etc. The UnitTest framework makes use of a quite specific syntax, which means that all AVA test cases will look quite similar. AVA provides an importable module called ava_api, which contains a list of targets, Context Simulators and a messaging subsystem.

- A Context Simulator emulates a subsystem that is not a part of specified test configuration. Context simulators are the binary files that simulates different part of the patient ventilator like monitoring, breathing, controlling
- The messaging subsystem provides a method to present messages in the terminal and in the generated test report.

Test cases in AVA follows the UnitTest test framework structure.

Tellus Patient Simulator
It runs as a separate application on an x86 Linux 32bit machine and communicates (asynchronously) via shared memory to the main breathing/monitoring applications. The main applications are only
aware of a special "FPGAComInterface", in the FPGA read/write code, and this interface needs to be enabled in the Breathing/Monitoring code during build [43].

5.2.15 Deployment

MBT application is a JavaFX application. It has been developed using JavaFX 2.2 which is fully integrated with the Java SE 7 Runtime Environment (JRE) and the Java Development Kit (JDK). JavaFX is currently deployed on Linux, Windows and Mac OS X.

5.2.16 Tools and frameworks

The following tools and frameworks have been used for application development:

- NetBeans IDE 7.3.1
- Eclipse Juno
- Java Development Kit (JDK) 7 update 25
- Python 2.7
- PyDev (Python IDE for Eclipse)

5.2.17 Self-Contained Application Packaging

JavaFX 2.2 adds new packaging option called Native Packaging, allowing packaging of an application as a "native bundle". This gives users a way to install and run an application without any external dependencies on a system JRE or FX SDK. Each self-contained application package includes the following:

- The application code, packaged into a set of JAR files, plus any other application resources (data files, native libraries)
- A private copy of the Java and JavaFX Runtimes, to be used by this application only
- A native launcher for the application
- Metadata, such as icons

The easiest way to produce a self-contained application is to modify the deployment task. To request creation of all applicable self-contained application packages simply add nativeBundles="all" to the <fx:deploy> task on build.xml.

Producing install packages for Linux assumes that the native tools needed to build install packages are installed. For RPM packages, this typically means the RPMBuild package and its dependencies. No admin permissions are needed to build the package.
After building the project using NetBeans IDE, the RPM package is created in the following directory of MBT project:

- MBT/dist/bundles

This directory will contain MBT-1.0-1.i686.rpm and MBT directory including all binaries required by the MBT application. After installing MBT-1.0-1.i686.rpm on a Linux machine, the application is installed on the local drive and runs as a standalone program using a private copy of Java and JavaFX runtimes. The application is launched in the same way as other native applications of operating system, for example by using a desktop shortcut or menu entry.

By default the application:

- Will install the application to /opt
- Will add a shortcut to the application menu
6 Quality Assurance

6.1 GUI Testing

Quality assurance is done in two phase. First phase is for ensuring compatibility and right behavior of GUI. After developing the GUI, our mission was to ensure that the GUI is compatible with different operating systems. The GUI is made in windows environment using JavaFX. Our target system runs only on Linux environment. So the compatibility testing was necessary. We tested our GUI in different Linux versions like Fedora, Ubuntu etc. After compatibility test different functionalities are tested. The tests are performed manually, no tool is used for GUI testing.

6.2 Model and Adapter Testing

The second stage of quality assurance is performed for the model and adapter. The model and adapter are tested thoroughly to make sure that both of these parts are error free and provide correct result in any circumstance. The whole system has a dataflow consists of several steps. It starts with the user interface that sends user data to its immediate combinatorics interface. Then combinatorics sends all combinations to the model one after another, model generates xml files which the adapter uses to send data to the SUT by protocol buffer. In all the stages of this flow tests are performed to check whether any part is facing incompatibility issues to accept data from its previous module. Lots of test suits are made using random figures throughout the limit of every parameter. The GUI is used for that. The test suits were xml files. Those test suits were run throughout whole night and output data was analyzed for every steps next day.

For three kinds of patient categories (Adult, Pediatric and Neonate), we have implemented 13 ventilation modes. With all the values of different parameters, different combinations are made using different criteria. For example, first combination is made with lower limit of all the parameters. Then in second combination one parameter is changed to higher limit and others are kept as it was before. Then another parameter is changed to higher limit and previous one again kept in lower limit. Then both of them are changed to high and others remained same. In this way the manual combinations are made for all parameters’ range. All the combinations are saved in the spreadsheets where for every mode, different spreadsheets are made. While test cases are made, the limit values were our main concern. Values were also chosen from outside the range to make input case with invalid value.
Then another test is performed. This time data is taken from the current version of patient ventilator (SERVO-U) [44]. Lots of valid data was taken as well as some invalid input for parameters are inserted manually in the spreadsheet. One column in the test case was filled with TRUE/FALSE using actual ventilator scenario. For both the case (range test and single value test) test suits and test cases are made from the newly made GUI for our MBT application. After making the test cases and test suits all these are scheduled to run through both the target system and model. This is also done using the MBT GUI. The outputs of the MBT application are XML and spreadsheet files. Output from the model and target system are compared to find the test result and improve the model iteratively.

In addition to the integration test [45] after the development of a single mode, the whole process described above is done in three phase, two are for limit values and one is for random values inside the ranges as well as outside. The result of these three phases is given below in tabular and graphical format:

### 6.2.1 Phase 1: Tabular Representation (Limit Value)

<table>
<thead>
<tr>
<th>Patient Category</th>
<th>Ventilation Mode</th>
<th>No. of test case</th>
<th>Correct Result</th>
<th>Wrong Result</th>
<th>Percentage Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adult</td>
<td>Pressure Control</td>
<td>3088</td>
<td>3088</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Adult</td>
<td>PRVC</td>
<td>1244</td>
<td>284</td>
<td>960</td>
<td>77.17%</td>
</tr>
<tr>
<td>Adult</td>
<td>Volume Control</td>
<td>1602</td>
<td>1319</td>
<td>283</td>
<td>17.67%</td>
</tr>
<tr>
<td>Pediatric</td>
<td>Pressure Control</td>
<td>1111</td>
<td>1111</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Patriotic</td>
<td>PRVC</td>
<td>1235</td>
<td>515</td>
<td>720</td>
<td>58.30%</td>
</tr>
<tr>
<td>Patriotic</td>
<td>Volume Control</td>
<td>1428</td>
<td>1232</td>
<td>196</td>
<td>13.73%</td>
</tr>
<tr>
<td>Neonatal</td>
<td>Pressure Control</td>
<td>956</td>
<td>956</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Neonatal</td>
<td>PRVC</td>
<td>1022</td>
<td>699</td>
<td>323</td>
<td>31.60%</td>
</tr>
</tbody>
</table>
6.2.2  Phase 1: Graphical Representation (Limit value)

![Percentage Error per Ventilation Mode (Phase 1)]

6.2.3  Phase 2: Tabular Representation (Limit Value)

<table>
<thead>
<tr>
<th>Patient Category</th>
<th>Ventilation Mode</th>
<th>No. of test case</th>
<th>Correct Result</th>
<th>Wrong Result</th>
<th>Percentage Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adult</td>
<td>Pressure Control</td>
<td>1012</td>
<td>1012</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Adult</td>
<td>PRVC</td>
<td>1244</td>
<td>1124</td>
<td>120</td>
<td>9.65%</td>
</tr>
<tr>
<td>Adult</td>
<td>Volume Control</td>
<td>1480</td>
<td>1424</td>
<td>56</td>
<td>3.78%</td>
</tr>
<tr>
<td>Pediatric</td>
<td>Pressure Control</td>
<td>1111</td>
<td>1111</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Patriotic</td>
<td>PRVC</td>
<td>1440</td>
<td>1351</td>
<td>89</td>
<td>6.18%</td>
</tr>
<tr>
<td>Patriotic</td>
<td>Volume Control</td>
<td>1238</td>
<td>1213</td>
<td>25</td>
<td>2.02%</td>
</tr>
<tr>
<td>Neonatal</td>
<td>Pressure Control</td>
<td>956</td>
<td>956</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Neonatal</td>
<td>PRVC</td>
<td>1124</td>
<td>1039</td>
<td>85</td>
<td>7.56%</td>
</tr>
</tbody>
</table>
6.2.4 Phase 2: Graphical Representation (Limit Value)

![Percentage Error per Ventilation Mode (Phase 2)](image)

6.2.5 Phase 3: Tabular Representation (Random Value)

<table>
<thead>
<tr>
<th>Patient Category</th>
<th>Ventilation Mode</th>
<th>No. of test case</th>
<th>Correct Result</th>
<th>Wrong Result</th>
<th>Percentage Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adult</td>
<td>Pressure Control</td>
<td>1020</td>
<td>1020</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Adult</td>
<td>PRVC</td>
<td>1320</td>
<td>1263</td>
<td>57</td>
<td>4.32%</td>
</tr>
<tr>
<td>Adult</td>
<td>Volume Control</td>
<td>1256</td>
<td>1216</td>
<td>40</td>
<td>3.18%</td>
</tr>
<tr>
<td>Pediatric</td>
<td>Pressure Control</td>
<td>956</td>
<td>956</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Patriotic</td>
<td>PRVC</td>
<td>1328</td>
<td>1291</td>
<td>37</td>
<td>2.79%</td>
</tr>
<tr>
<td>Patriotic</td>
<td>Volume Control</td>
<td>1238</td>
<td>1213</td>
<td>25</td>
<td>2.02%</td>
</tr>
<tr>
<td>Neonatal</td>
<td>Pressure Control</td>
<td>850</td>
<td>850</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Neonatal</td>
<td>PRVC</td>
<td>1124</td>
<td>1039</td>
<td>85</td>
<td>7.56%</td>
</tr>
</tbody>
</table>
6.2.6 Phase 3: Graphical Representation (Random Value)

From the tabular and graphical representation of the test results, it is noticeable that in the initial phase, the percentage errors were significantly high. After the initial phase, the output is used as feedback to improve the model again. Therefore, in the next two phases the error percentage has become very low, which is below 10%. However, still some of the error remains. The errors occurred because of using different version of specification document than SUT. We used the updated documents for modeling whereas the SUT (the patient simulator) was made following documents of the older version of patient ventilator. Every parameter in the ventilator has a default value which is loaded according to the patient category and ventilation mode. Some of the default values were different in different versions. In addition, for volume control and PRVC ventilation mode, the specification of the parameter Tidal Volume is changed. It was also a reason for few errors. It is noted that in the table above we have mentioned 9 ventilation modes though we have implemented 13. It is because the other modes are more or less same with the mode mentioned in the table. That is why those modes are tested with very insignificant number of test cases. It is also noted that for neonatal patient category, there is no column for volume control mode. This is because the volume control mode is not implemented in the actual machine for this patient category.
Limitation and Future work

For making model no commercial or open source tool is used though idea of implementing model based testing was taken from those tools. The model was made with plain java programming language so any modification or upgradation of the model needs the expertise of knowledge in java. The tool that we have built as output of our work is only suitable for that particular product. It cannot be used in other extents.

The requirement specification documentation that provided for modeling and the real implementation of SERVO-I (current patient ventilator product) is different. So the model showed some differences with the real implementation. Same problems occurred while working with the simulator. It is very important to note that the default values can be distinctive in distinctive version of the patient simulator. It will affect the model output in some extents, even there is a possibility that for the two same inputs in different version of the simulator one can be failed and one can be passed.

We only implemented the ventilation parameters in our model but we did not take the alarm parameters in our consideration. It is possible to extend the horizon of the tool by implementing alarm section of the system.

Even though the GUI is compatible in different platforms (Windows, Linux, Mac OS), it is not possible to run whole system without Linux because the existing test framework (AVA + HEVT) runs on Linux environment. The SUT (patient simulator) is also a combination of some binary files which runs only on Linux. It will be a good endeavor to make the whole system platform independent or at least the test framework. So even if the SUT runs on Linux environment, it will be possible to send/receive data to the target system remotely.
8 Conclusion

This project contributes to the organization to improve their existing Exploratory or Session-Based Testing framework to Model-based testing system. Our project complements their existing framework so that instead of a particular value or limit value now they can run all the values in the range. Examples are given in the introduction that proves the fact that for such life-critical equipment it is not wise to test only limit value to ensure the system will work for the whole range. Therefore, one important mission of this project was testing all possible values in the range instead of limit values. The second mission was to make a satisfactory coverage to ensure sufficient system and subsystem testing. The third mission was to cover all the possible combinations of parameters’ value that can be run under the system and subsystem tests. In this project, the above missions were successful by implementing model-based testing. For the output from this project, a GUI is made for handling all the selection of patient categories, ventilation modes, parameter range, unit and value selection, number of maximum test case selections and other utility controlling. Our customized combinatorics is used for selecting all the possible combinations of parameters. Model works as an engine to make a decision for parameters' validity and to provide oracle information. An adapter is an intermediate system for sending of model output and receiving output from target system. The output is presented in two ways that are with xml and spreadsheet. Xml files are used by the GUI to present to the user in a comprehensive way. Spreadsheets also provide detail information that helps for quality assurance of the application. In summary, with combination of our implementation and existing test framework a fairly complete MBT tool is created for the company.
Bibliography


