



TAMPERE UNIVERSITY OF TECHNOLOGY

JANNE SJÖGREN

AUTOMATED TESTING OF SOLAR INVERTER SOFTWARE

Master of Science Thesis

Examiner: Professor Teuvo Suntio  
Examiner and topic approved by the  
Faculty Council of the Faculty of  
Automation, Mechanical and  
Materials Engineering on 6 April  
2011.

## ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY

Master's Degree Programme in Automation Technology

**SJÖGREN, JANNE:** Automated Testing of Solar Inverter Software

Master of Science Thesis, 56 pages

December 2011

Major: Power Electronics of Electrical Drives

Examiner: Professor Teuvo Suntio

Keywords: Automated testing, solar inverter, software testing

Testing is a necessary process for verifying correct functioning of a product. When testing is performed manually, it consumes a lot of time and labor. In this thesis, an automated testing system for a solar inverter is designed and implemented into a software development team. The objective was to make testing more effective and release testers from simple and monotonous tasks to more demanding ones. At the same time, human mistakes in testing can be avoided.

Options for the implementation were searched from literacy and Internet. Also implementations inside the organization were considered. The system was realized into the development team in tight time constraints leaving space for further expansion of the system.

The beginning of the thesis concentrates on literature research where theories on software testing and on automation system are presented. Additionally, the characteristics of agile methods and their applicability to the automation system are discussed. In the end of the thesis, the design of the system is presented by explaining how different parts of the system were executed. Finally, the realizations of the most fundamental tests are presented as well as future prospects are discussed.

The resulting automated testing system not only facilitates the testing process but also enables having always the latest up-to-date software under version control. In this manner, the software development team is always aware of the state of the product, which helps in avoiding massive integration work just before a large release by solving integration problems all along the development process. This also enables having the latest software version always ready to be tested.

## TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO

Automaatiotekniikan koulutusohjelma

**SJÖGREN, JANNE:** Aurinkosähkötaajuusmuuttajan ohjelmiston automaattinen testaus

Diplomityö, 56 sivua

Joulukuu 2011

Pääaine: Sähkökäyttöjen tehoelektroniikka

Tarkastaja: Professori Teuvo Suntio

Avainsanat: Automaattinen testaus, aurinkosähkötaajuusmuuttaja, ohjelmiston testaus

Testaus on välttämätön prosessi tuotteen oikean toiminnallisuuden varmistamiseksi. Kun testaus suoritetaan manuaalisesti, se kuluttaa paljon aikaa ja työvoimaa. Tässä työssä suunnitellaan ja toteutetaan ohjelmiston tuotekehitystiimiin aurinkosähkötaajuusmuuttajan ohjelmiston automaattinen testausjärjestelmä. Työn tavoite oli tehdä testauksesta tehokkaampaa ja vapauttaa testaajat yksinkertaisesta ja monotonisesta työtehtävästä vaativampiin työtehtäviin. Samalla vältetään inhimillisiltä testausvirheiltä.

Toteutusvaihtoehtoja tutkittiin kirjallisuudesta ja internetistä. Myös organisaation sisäisiä toteutusmenetelmiä tarkasteltiin. Järjestelmä toteutettiin tuotekehitystiimiin tiiviissä aikataulussa lähtien liikkeelle aivan alusta mahdollistaen myös myöhemmin lisättävät järjestelmän laajennukset.

Työn alussa sijaitsevassa kirjallisuustutkimuksessa käsitellään ohjelmistotestauksen ja luodun automaatiojärjestelmän teoriaa. Lisäksi käsitellään ketterien järjestelmien ominaisuuksia ja niiden soveltuvuutta suunniteltuun automaatiojärjestelmään. Työn loppupuolella esitetään järjestelmän eri osien suunnittelu ja toteutus. Viimeiseksi esitetään olennaisimpien testien toteutus ja tarkastellaan järjestelmän tulevaisuuden näkymiä.

Työn tuloksena toteutettu automaattinen testausjärjestelmä ei ainoastaan helpota testausprosessia, vaan se myös mahdollistaa sen, että viimeisin päivitetty ohjelmistoversio on aina versiohallinnassa. Ratkaisemalla integraatio-ongelmat jo tuotekehitysprosessin aikana ohjelmiston kehitystiimi pysyy aina ajan tasalla tuotteen tilasta, mikä auttaa välttämään massiivista juuri ennen suurta julkaisua tapahtuvaa integraatiotyötä. Täten viimeisin ohjelmistoversio on aina saatavilla valmiina testattavaksi.

## PREFACE

The topic of the thesis was provided by ABB Drives. The thesis was examined by Professor Teuvo Suntio from Tampere University of Technology. The supervisor at ABB Drives was M.Sc. Terho Läärä.

I want to address my gradidute to Terho for the support and guiding. Also, I want to thank all the people at ABB Drives who provided me with any supportive information concerning this thesis. Furthermore, I want to thank Teuvo for the advices and for examining this thesis.

Helsinki, November 1, 2011

Janne Sjögren

# CONTENTS

1	Introduction .....	1
2	Software testing.....	3
2.1	Introduction .....	3
2.2	Testing techniques.....	4
2.3	Software development process.....	6
2.4	Test-driven development.....	9
2.5	Regression testing .....	10
3	Automation system .....	11
3.1	Embedded system.....	11
3.2	Solar inverter.....	13
3.2.1	Maximum power point tracking.....	14
3.3	Automated testing process .....	15
3.4	Test automation framework .....	15
3.4.1	Automated tester framework.....	16
3.4.2	Visual Studio.....	17
4	Agile methods .....	19
4.1	Continuous integration .....	20
4.1.1	The value of continuous integration.....	23
4.1.2	Build script .....	24
4.1.3	Continuous integration server .....	25
4.1.4	Continuous integration tool.....	25
4.1.5	Crucial practices.....	26
4.2	Version control.....	27
4.2.1	Git.....	28
5	Design of the testing system .....	30
5.1	Introduction .....	30
5.2	The set-up.....	31
5.3	Continuous integration server .....	32
5.4	Local workstations .....	34
5.5	Running the builds .....	35
5.6	Feedback .....	36
5.7	Backups .....	38
6	Test realisations.....	39
6.1	Introduction .....	39
6.2	Grid simulation tests .....	40
6.3	Batch scripts .....	43
7	Future prospects .....	45
8	Summary .....	47
	Sources .....	48

## TERMS AND DEFINITIONS

ABB	Asea Brown Boveri, multinational corporation operating mainly in power and automation industry
AC	Alternating Current
A/D	From Analog to Digital
ASIC	Application-Specific Integrated Circuit
ATF	Automated Tester Framework
C	A programming language
C++	A programming language
C#	C sharp, a programming language
Code refactoring	Altering the structure of a code without changing its external behavior
CPU	Central Processing Unit
D/A	From Digital to Analog
DC	Direct Current
EMI	Electromagnetic Interference
FPGA	Field-Programmable Gate Array
HTML	Hypertext Markup Language, a markup language for web pages
IEEE	Institute of Electrical and Electronics Engineers, association that is dedicated to advancement of technology
MHTML	An archive format
PC	Personal Computer
Sinus filter	A filter for reducing peak voltages and currents
SMS	Short Message Service, text messaging service for mobile or web communication
TDD	Test-Driven Development, a software development process
USB	Universal Serial Bus, a standard for connection between computers and electrical devices
VB.NET	Visual Basic .NET, a programming language
XML	Extensible Markup Language, a set of rules for encoding documents
.NET Framework	A software framework

## SYMBOLS

$I_{MP}$	Current in maximum power point
$P_{MAX}$	Maximum power
$V_{MP}$	Voltage in maximum power point
$V_{OC}$	Open circuit voltage

# 1 INTRODUCTION

Automation system is an embedded system that consists of the hardware, the software and the connections to the environment of the system. As the systems have become larger, also the software has become larger and more complicated. Testing of the large software takes a huge amount of time, which leads to longer product development times. In order to succeed in the competition between the competitors, the product development has to be fast to get the product to the market as soon as possible. Several studies indicate that testing can take even a half of the product development phase. Proper automated testing of the software enables significant savings and competitive advantages.

No matter how carefully software is believed to be generated, bugs will always exist due to human mistakes. It is rather a rule than an exception that comprehensive testing of the software requires more time than it was allocated in the development process. This causes temporal and financial challenges to development teams achieving to meet deadlines generated in the beginning of development processes. When problems occur in the process, it often causes delay in delivering the product to customers, leading to a point, where software developers have to work extra hours or more employees have to be hired in order to meet upcoming time limits. This is when discussion on creating a competent automated software testing environment for testing a product becomes topical.

The idea in automation of tests is to release testers from tedious testing tasks as well as getting rid of human testing mistakes. Before performing this thesis, the software tests were used to be executed manually by a couple of testers who were working in a laboratory over a weekend executing tests from a certain list. By automating the tests, developers are able to run the set of tests automatically whenever they want: continuously, periodically or in both manners. The tests can be run also nightly, so that test reports are available in the morning when developers come to work.

The design and the implementation of automation tests take time. Yet, a lot of time can be saved in long product development processes due to the repetitive nature of testing. As the tests are run daily, errors can be found early. The earlier errors in the code are discovered, the easier they are to be fixed. Conversely, without regularly run automated tests, the integration phase in the end of the development cycle can expand to massive dimensions. When huge problems are discovered in the late integration phase, the delivery of the program can be significantly delayed. Sometimes the project can even run to a dead end.

This thesis is focused on designing and implementing an automated software testing system for a solar inverter in order to make testing more effective. The development process of the solar inverter was already in the maintenance phase when this thesis was introduced. Hence, the objective was to create an automated testing environment on system testing level for ensuring that malfunctions would not occur as new software updates are released.

In the beginning of the thesis, theories of software testing as well as parts of the automation system are introduced. Further, discussion on agile methods and its advantages for the system take place. Especially possibilities provided by continuous integration and distributed version control are explained. This is followed by presentation of the designed system where the set-up and its parts are being introduced. In the end, some of the generated scripts are presented and explained. In addition, future prospects and development possibilities are discussed.



## 2 SOFTWARE TESTING

This chapter concentrates on defining what is software testing, what for it is utilized, and what kinds of testing techniques exist. Also, software development processes such as Waterfall-model, V-model, and Test-driven development are introduced. Regression testing, which is of high importance concerning the existence of this thesis, is presented in the end of the chapter.

### 2.1 Introduction

Software testing is used for achieving empirical information on the quality of software in an environment in which the software is planned to work. The objective of testing is to find possible errors in the code by running specified tests. In this manner, the requirements, which determine that the software is working as expected, can be verified.

Error is a deviation from the specifications of a product, the origins of which derive usually from a human mistake. Despite of testing, some errors remain in the software of which some may remain undiscovered. Defects can be caused to a system by defective execution of a part of the program. They can be fixed by another fault or functionality, but they can also cause a visible failure in the system. [1]

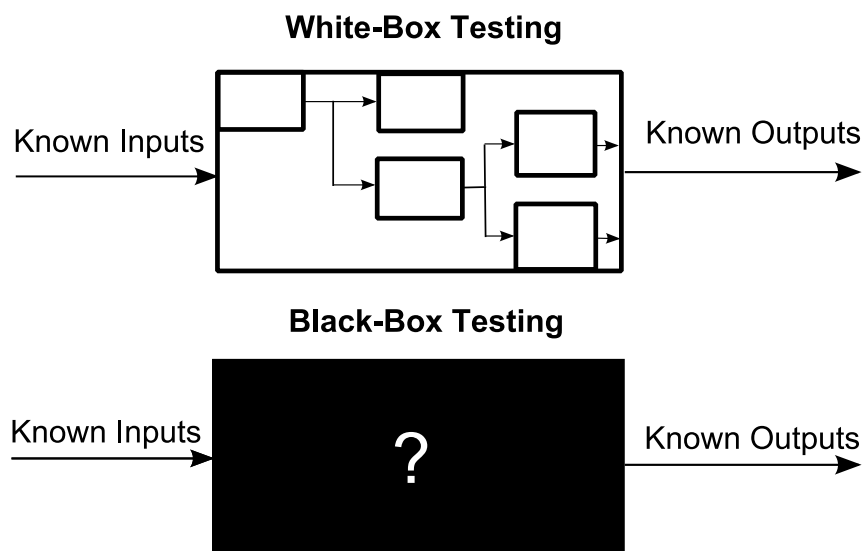
Testing is not the same as debugging. Testing is for pointing out errors by using specifications, so that the starting point and the correct result are known in advance. Debugging is for locating an error or misconception that led to the failure of the program and then fixing it, without knowing the starting point or the correct result. Debugging demands specific knowledge about the content of the program. Therefore, debugging cannot be performed by an outsider unlike testing, which is performed according to specifications. Debugging cannot be automated. Usually, debugging follows testing. [2]

Software testing provides information to the organization about how to proceed. Testing itself does not directly enhance the quality of the system. Nevertheless, by offering an aspect to detected weaknesses, it provides the organization an opportunity to make conscious decisions on focusing recourses for improving quality of the system. [3]

Software testing is performed on three levels. First of the levels is unit testing, in which the correct functioning of the software is verified at unit or component level. Units can be, for instance, individual functions or programs. The programmers themselves often perform the unit tests. The second level is integration testing, where separate units are united for verifying their correct interaction between each other. Depending on the size of the project, the programmer or an independent tester performs

the integration tests. The third level is system testing, in which complete, integrated software is tested. The purpose is to evaluate compliance of the system with the specified requirements instead of structural testing. Normally, the programmer or an independent tester performs the system tests. [22; 23; 24]

The two first levels are performed using white-box testing (figure 2.1), meaning that testing the internals of the software, such as fixing syntax and logical errors, is the main focus. In white-box testing, only the structure of the software is known, leaving unknown the operation of the program. Despite the types of tests that are executed, their intent is to verify that the system successfully satisfies entirely its functional requirements. Black-box testing is applied on the third level, which means that tester is aware of the entire operation of the system without knowing the structure of the software. It performs most parts of the system by invoking various system calls through user interface interaction. The focus in this thesis is the system testing. [6]



*Figure 2.1. White-Box and Black-Box Testing.*

## 2.2 Testing techniques

In order to verify that a program is functioning properly, all the possible inputs and paths should be tested to perfection. Practically, this is an impossible task to perform. For instance, two figures from between  $-2^{15}$  and  $2^{15}-1$  are added together. Therefore, the amount of possible inputs is more than four millions. In practice, testing a program by providing more than four million inputs at a time is not possible. In addition, different states, indeterminateness of the real-time systems, loops, and conditions have to be considered. Therefore, implicit testing of a program is not possible. [1]

Although programs cannot be tested implicitly, they can be tested soundly and carefully. By choosing reasonable test data, defects of the code can be found with good probability. Different software testing methodologies exist which are broadly divided into static techniques and dynamic techniques. [20]

Static software techniques perform testing of a component without execution of the software. It is carried out via static analysis of the code. One of the most powerful static techniques is review, which can be performed formally or informally. Walkthrough, technical review, and inspection are the types of reviewing techniques. In walkthrough technique, the author of the document to be reviewed guides the participants through the document and gathers feedback. Technical review is a peer group discussion where the object is to achieve consensus on the technical approach taken, while developing the system. Inspection is also a type of peer review with the focus in the visual examination to detect any defect in the system. In addition, static analysis tools exist, which inspect coding standards, code metrics, and code structure. [20]

In dynamic software techniques, the code is tested for finding defects. The technique is divided into three categories: specification based technique, structure based technique, and experience based technique. Specification based technique is also known as black-box testing. There are five main specification based testing techniques: equivalence partitioning, boundary value analysis, decision table, state transition testing, and use case testing. [20]

In equivalence partitioning, the test cases are designed so that the test cases cover every partition at least once. The idea is to divide test conditions into sub groups, which can be considered the same. For instance, if 1 to 100 are the valid values, valid partitioning is 1 to 50, and 50 to 100. Therefore, 1, 50 and 100 are the values for which the system has to be checked. Additionally, invalid partitions, such as random values -5 and 120 outside the boundaries, have to be checked. [20]

Boundary value analysis concentrates on testing input or output values on the edge of an equivalence partition or at the smallest incremental distance on either side of an edge. Boundaries are tested between the partitions for both valid and invalid boundaries. For instance, if the valid inputs are 1 to 99, then the test cases should be designed to include values 0, 1, 99 and 100 in order to verify the functionality of the system. [20]

Decision table is also known as cause-effect table. The table contains a combination of inputs with their associated outputs used to design test cases. The first task is to identify a suitable function, which has functional traits that react according to a combination of inputs. If there are two conditions, there are four combinations of input sets. If there are three conditions, there are eight combinations correspondingly. [20]

In state transition testing, any aspect of the component can be described as a finite state machine. The test cases are designed to execute valid and invalid state transition. In any given state, one event can give rise to only one action, but the same event from another state may cause different action and a different end state. In use case testing, the test cases are designed to execute real life scenarios by identifying test cases that exercise the whole system on a transaction-by-transaction basis from the beginning to the end. These cases help to resolve integration defects. [20]

Structure based testing is also known as white-box-testing. Test coverage measurements and structural test case design are the two purposes of the structure based testing techniques. Test coverage, statement coverage and statement testing, and

decision coverage and decision testing are the most utilized techniques. Test coverage describes the percentage, to which a certain coverage item has been exercised by a test suite. Statement coverage and statement testing present the percentage of the executable statements, which have been exercised by a test suite. Decision coverage and decision testing count the statements exercised like, among others, ‘if-statements’, loop-statements, and case statements. [20]

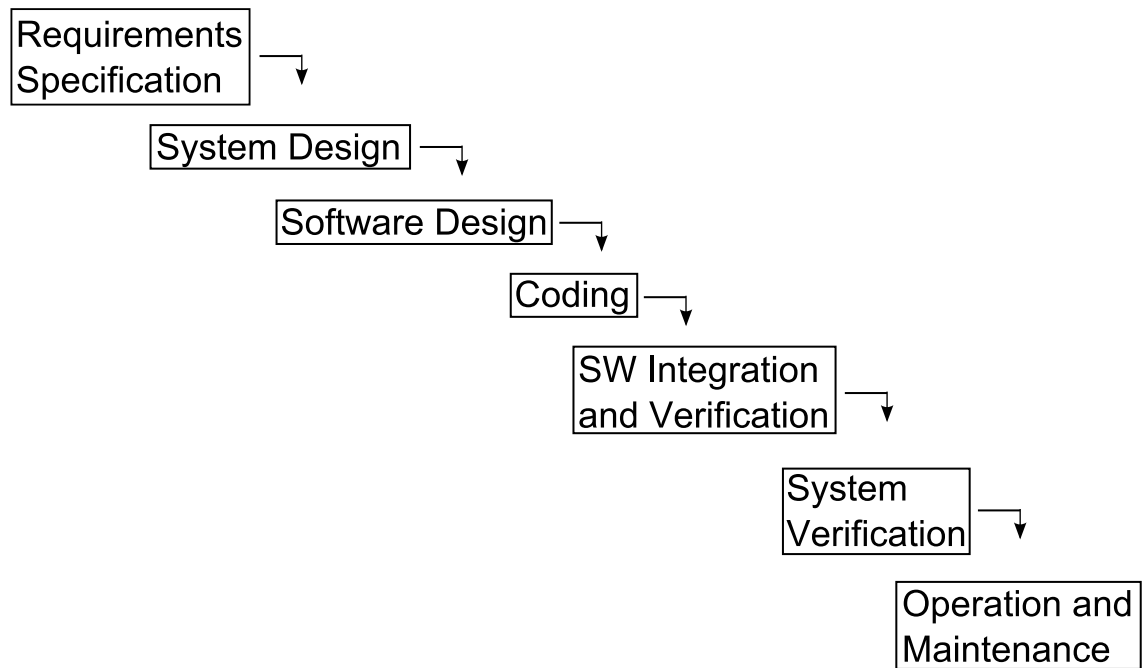
Experience based testing is based on knowledge, experience, intuition or imagination of a person. An experienced tester is often able to locate an elusive defect in the system. Error guessing and exploratory testing are the techniques in this category. In error guessing technique, the experience of a tester is tested to seek for elusive bugs, which may be a part of a component or a system. This technique is often brought into use after the formal techniques have been utilized. A structured approach is to list possible defects and then try to reproduce them via test cases. [20]

Exploratory testing is also known as ‘monkey testing’, where minimum planning and maximum testing take place as a hands-on approach. The test execution and test design happen simultaneously without formally documenting test conditions, test cases or test scripts. The approach is useful, when time at hand is exceptionally limited or the project specifications are poor. [20]

Testing techniques to be utilized depend on multiple factors. The main factors are urgency and severity of the project, as well as available resources. Not all the techniques are utilized at a time in all the projects. Decisions concerning the techniques utilized depend on organizational policies. [20]

## **2.3 Software development process**

Testing is a part of the software development process. Several studies state that the testing phase can take even a half of the time of the development process. Traditional way of describing software development process is the waterfall model (Fig. 2.2).



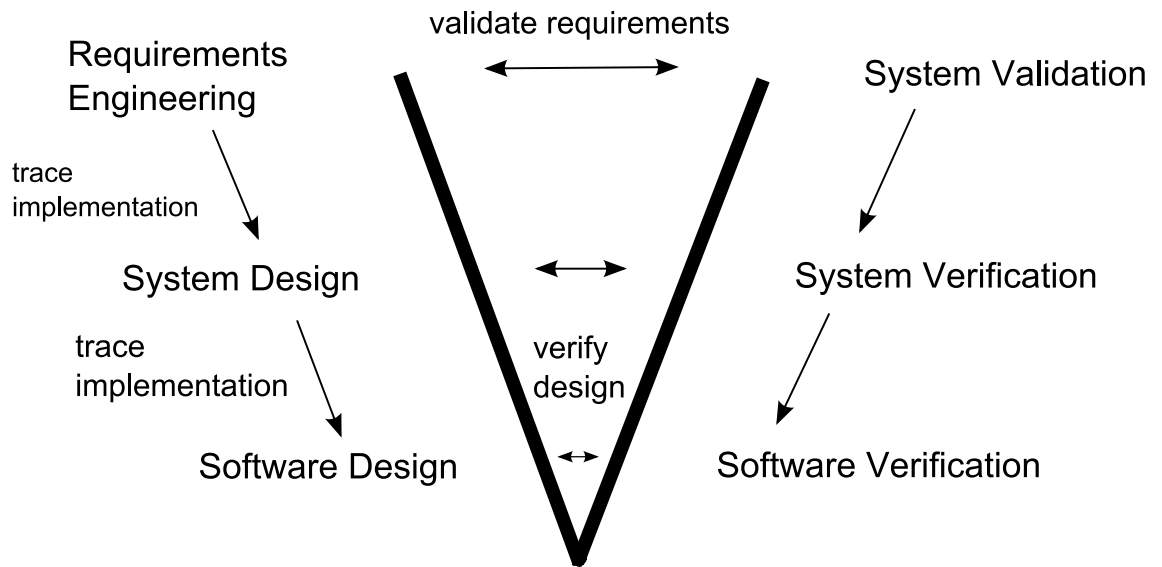
**Figure 2.2.** Waterfall Model [27]

At first, all the requirements of the system that have to be developed are analyzed. The second step is to design the system properly before starting implementation. This phase contains the architectural design by defining the main blocks and components of the system, as well as their interfaces and interactions. The third step is the software design, which is based on the system architecture. In this phase, the software blocks are defined into code modules. A software design document, which is the base of the following implementation work, is the output of this phase. Next step is coding, where the system is developed in smaller portions, called units, which are able to stand alone and are integrated later to form the complete software package. The fifth step is software integrations and verification. This phase involves unit tests and integration tests. After successful integration tests, the complete system is tested against the initial requirements. In the last phase, the system is handed over to the customer. Possible modifications to the software are made to meet the demands of the customer. Often this phase is extended to a never-ending phase as the customer discovers new shortages in the software. [27]

Waterfall method, however, has its weaknesses. It is often difficult to gather all possible requirements during the first phase. If not, the subsequent phases will suffer from it. Additionally, iterations are meant to happen within the same phase. The problems commonly tend to shift to the later phases, which eventually results in a bad system design. Thereby, instead of solving the root causes the tendency is to patch problems with inadequate measures. Furthermore, a large maintenance phase might result as the further development is squeezed into the end. [27]

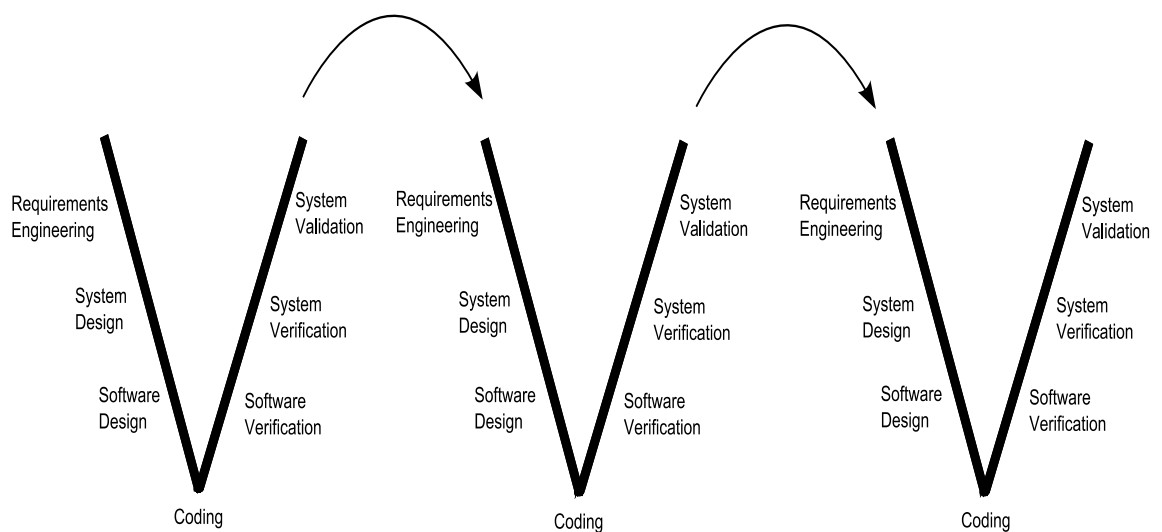
A more developed model of the waterfall model is called V-model in which the individual steps are almost the same. The V-shape forms as the process steps are bent upwards at the coding phase because the design phases and testing phases are related to

each other as shown in Fig. 2.3. The tests are derived from their design or requirements counterparts. This correlation allows verifying each of the design steps individually. [27]



**Figure 2.3.** V-Model.[27]

Not only the correct implements of requirements have to be checked but also if the requirements are correct. When the requirements have to be updated, subsequently the design and the coding have to be updated. This has to be treated either in a never-ending maintenance phase in the waterfall model, or in going over to another V-cycle. This kind of Multi-V-model is shown in Fig. 2.4.



**Figure 2.4.** Multi-V-model. [27]

## 2.4 Test-driven development

Test-driven development (TDD) is an agile software development strategy that addresses both design and testing [12]. The fundamental programming theory is that the tests are written prior to the source code. The tests are added gradually during the implementation process. As the tests are passed, the code is refactored to improve the internal structure. The cycle continues until all the functionality is implemented. [21]

Although, TDD philosophy has existed since the 1960s, it has emerged recently as a novel software development approach to provide software of good quality [21]. TDD is considered as an essential strategy in such an emergent design because when writing a test prior to code, the developer deliberates and decides not only on the interface of the software, but also on the behavior of the software. Program 2.1 provides an example of a test written in C#. [12]

```
public void testCreateEmptyWallet()
{
    Wallet a = new Wallet();
    Assert.AreEqual(0, a.getNumMoney() );
}
```

**Program 2.1.** *A simple test for verifying that Wallet has zero money.*

Albeit the test is simple, it involves several design decisions including the class name (Wallet), the expectations of a default constructor, a new method (getNumMoney()) that returns an integer, and the expectation that a default Wallet has no money. In this manner, contemplating and writing tests before the actual code, better quality of software ought to be reached. [12]

Typically in TDD philosophy, developers produce code, debug, and produce some more code. When testers find a defect in the code, they do not only remove the defect, but also write a test that reproduces it. In this manner, the same defect will be also discovered in the future. [19]

Both the practice and literature indicate that the utilization of TDD yields several benefits. TDD leads to improved test coverage and simplifies the design by producing highly cohesive and loosely coupled systems. TDD also enables implementation scope to be more explicit. In addition, TDD improves the code quality by identifying likely breaking points early [19]. As a positive side effect, TDD may enhance job satisfaction and confidence. Also, larger teams of developers can work on the same code base due to more frequent integration. [21]

Nevertheless, some contradiction against the virtues of TDD exists. Many researches on the subject have been performed and not all of them agree with the claims in the literature, especially, when the TDD users are inexperienced developers. On the other hand, as a positive contribution, TDD proves to improve test coverage. According to some literature, TDD is the only means to achieve excellent test coverage [11]. [21]

## 2.5 Regression testing

Regression testing is quality control measure to ensure that the modified code still complies with the specified requirements and the unmodified code has not been affected by the maintenance activity. Regression testing is used for testing the impact of changes made in the code. Testers should obtain the input from the development team about the nature of the fix so that testers can check the fix first and then the effects of the fix. In regression testing, the test cases are run multiple times on different builds. Test cases include all test cases considered important for the functionality of the program. The objective of retesting the modified software is to ensure that all the bugs have been fixed and all the determined functionality is still working. [8]

In this implementation, regression tests are executed on the system level. System tests encompass a complete software system and therefore require a fully installed system. These tests verify that external interfaces work together as designed. System tests tend to have lengthy runtimes and therefore their schedules demand planning. In this implementation, system tests are run each time a new update is committed to mainline of the repository, since the development team consists of only few developers and the set of tests is fast to run. System tests are often run in intervals in larger development teams with lengthy set of tests to avoid prolonged queues. [7]



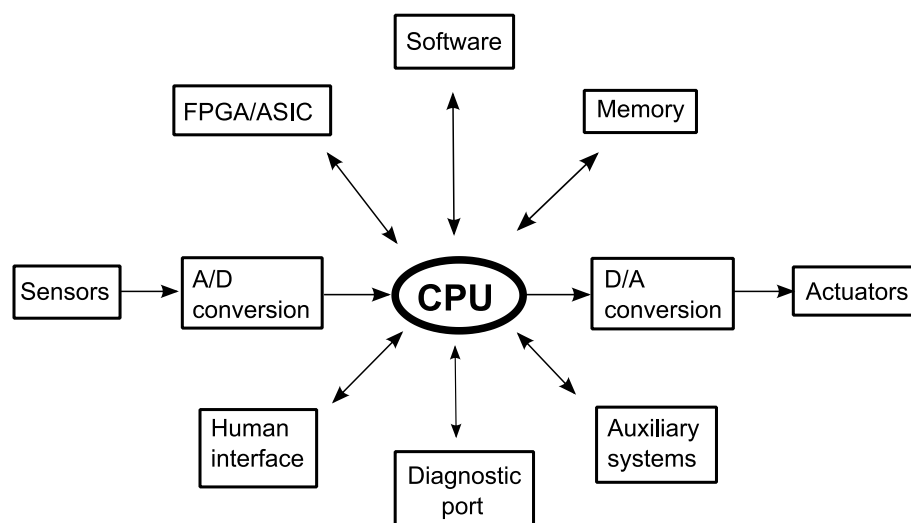
### 3 AUTOMATION SYSTEM

In this chapter, the idea is to outline the image of the automation system and the theory behind it. As a solar inverter is an embedded system, the concept of embedded system is clarified at first. Next, the basic functionality of a solar inverter is introduced, and the main idea of the content of the automation system is described. Finally, the concept of test automation framework is clarified.

#### 3.1 Embedded system

Embedded system is a microcontroller based device or software, which is designed to execute certain task with different choices and options. It is not designed to be programmed by the end user. Embedded system is a generic term for a broad variety of systems, such as cellular phone, navigator, missile tracking system and solar inverter. [3; 10]

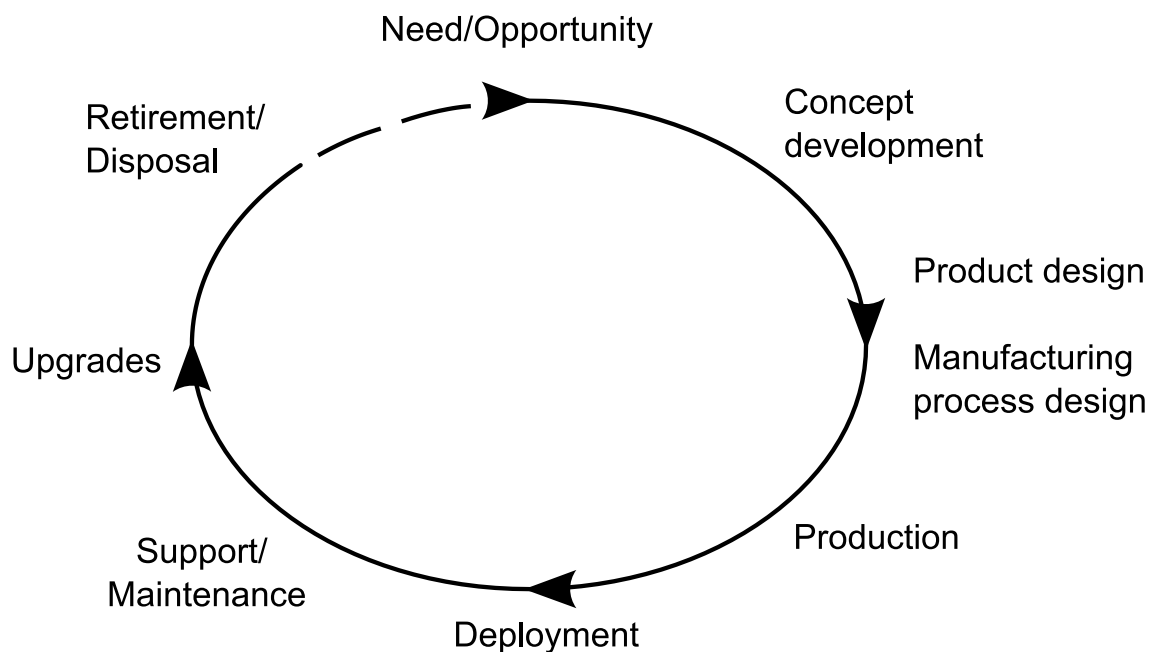
A processor performs the designed tasks of the system. The software of embedded system is loaded from non-volatile memory as the system is started. In simple systems the software consists of few functions spinning in loops. In more complex systems, such as a solar inverter, scheduling is also needed. Scheduling is actualized by a scheduler or an operating system, which controls the execution of the program. In real-time systems the system has to react to certain events or inputs in definite time. Operating system is in interaction with the real world via sensors and actuators. It has to react to the inputs and events within certain time constraints of its environment. Figure 3.1 describes an embedded system.



*Figure 3.1. Embedded system. [13]*

Embedded systems provide functionality specific to their application. Typically they execute control laws, signal processing algorithms, and finite state machines instead of executing, for example, word processing or engineering analysis. They must often detect and react to faults in both computing and surrounding electromechanical systems, as they must manipulate application-specific user interface devices. [13]

Figure 3.2 describes the lifecycle of an embedded system. At first, a need or an opportunity to deploy new technology is identified. Then a product concept is developed including analysis of the market trends as well as brainstorming of innovative ideas driven by technology trends and customer demands. This is followed by parallel product and manufacturing process design, production, and deployment. In order to create a profitable design, the designer must see past deployment and take into account support, maintenance, upgrades, and system retirement issues. [13]



**Figure 3.2.** *Lifecycle of an embedded system.* [13]

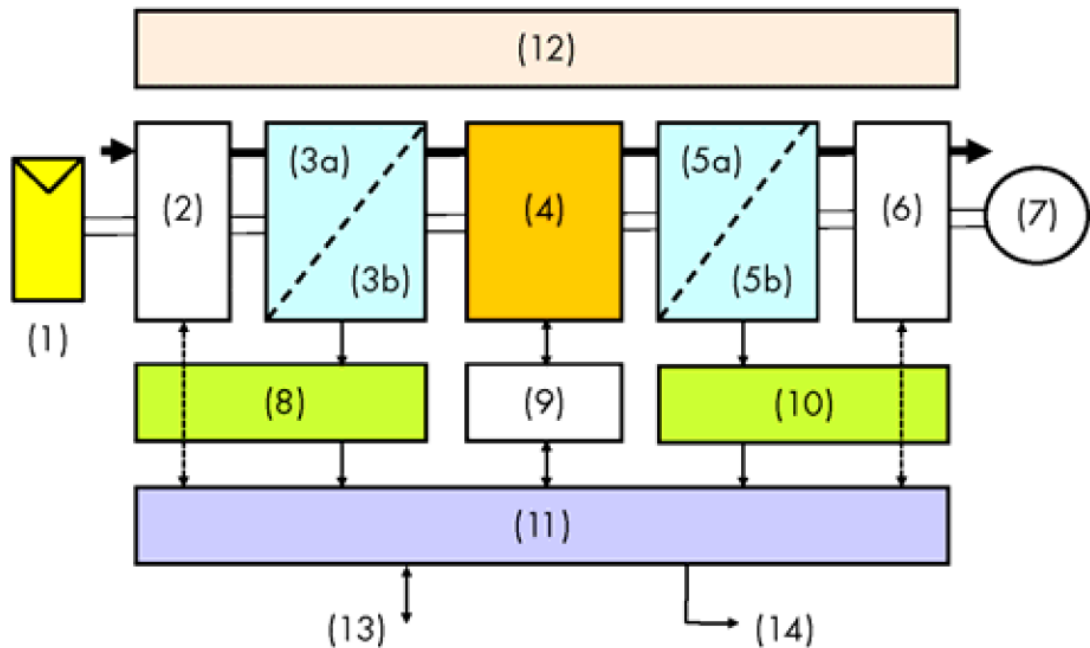
Along with the embedded system lifecycle, software testing development should advance side by side with the product design and development all the way until the software is no more developed. In this thesis, automated software testing system was implemented in the maintenance phase of the project, which is an advantage not only for the project, but also especially for the upcoming projects.

There are several means of testing software. In this thesis, the system was designed by exploiting the use of agile methods, more precisely, continuous integration environment, which enables the software to be tested automatically and frequently, so that errors can be found as quickly as possible. More about continuous integration is discussed later in this thesis.

### 3.2 Solar inverter

Solar inverter is used to convert DC, created by the solar generator, into AC. Most of the solar inverters are grid connected so they are configured to feed AC power to the grid. Modern solar inverters include additional functions like DC or AC parameter measurements, controlling solar generator, communication to user or grid and monitoring and protection of the complete solar system. [15]

Figure 3.3 describes functions of a solar inverter in a bloc diagram. Power semiconductors (see bloc (4)) in a suitable topology constitute the heart of an inverter. Between power semiconductors and solar generator (1) are DC protection and switches (2) followed by EMI (3a) and DC filter (3b). At AC side EMI (5a) and Sinus filter (5b), as well as AC protection and switches (6) to the grid (7) are located. Measuring equipment for electrical parameters on DC and AC side ((8) and (10)) and drivers (9) interact between controller units and hardware components. The controller (11) acts as pulse pattern generator and protection, controlling, monitoring and interfacing unit. Dissipating components like semiconductors are connected thermally to a cooling unit, which is connected to the housing (12). [15]



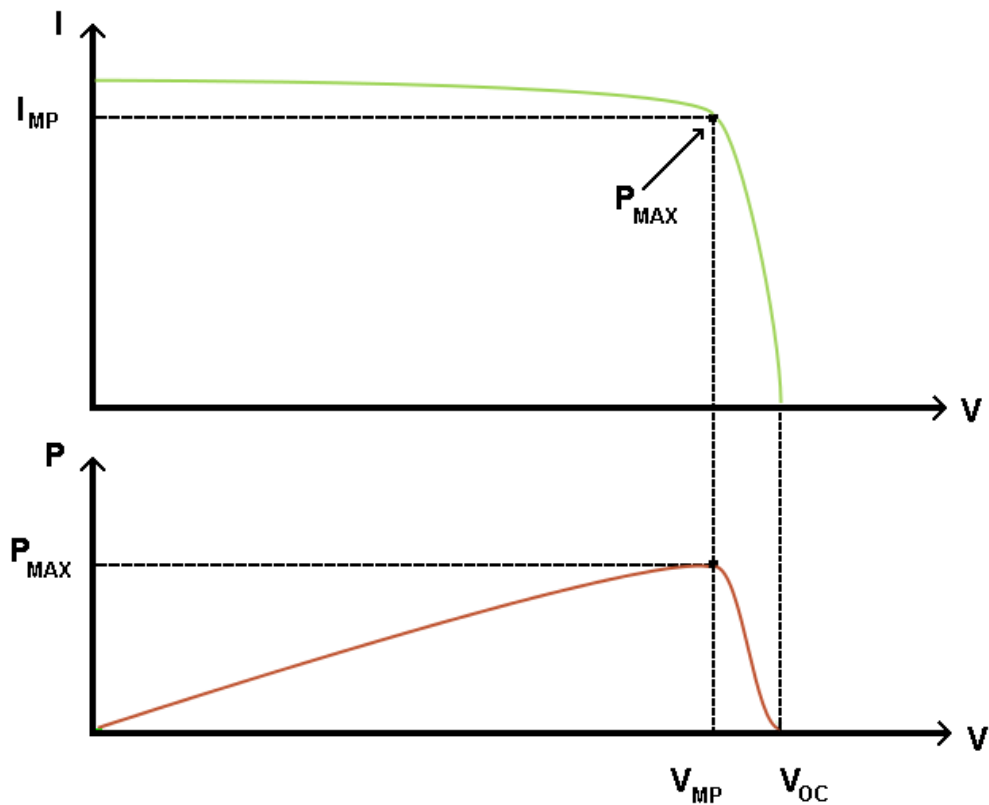
**Figure 3.3.** Bloc diagram of a solar inverter. [15]

The most important indicators for characterizing advances in inverter technology are inverter costs, efficiency and losses, as well as reliability and service. By automating software testing, an improving impact on reliability is achieved as faults are detected as they occur and they are easy to repair. This is very crucial since also the safety system

has to work in every circumstance. The system has to be shut down safely when critical faults occur, without causing detriment to the environment. [15]

### 3.2.1 Maximum power point tracking

Maximum power point tracker is one of the most fundamental functions of a solar inverter. The purpose of the tracker is to run the inverter into a point in which the power of the solar panel and the power fed to the grid are at the maximum. Several different algorithms for this purpose exist. Typically, they utilize at least measurements of the voltage of the solar panel as well as measurements of the current produced by the inverter to track the maximum power point. Additionally, depending on the algorithm, the current of the solar panel can also be measured. Figure 3.4 illustrates where the maximum power point is located.



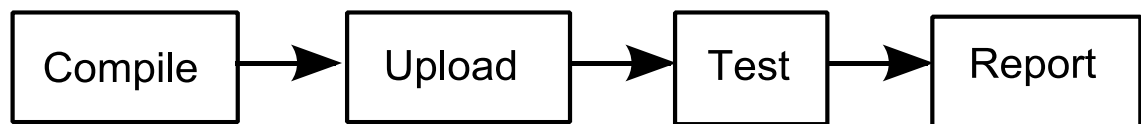
**Figure 3.4.** Maximum power point. [18]

As the intensity of the sunlight varies, also the power of the solar panel changes as well as the power of the solar inverter. The same power can be achieved with different values of current and voltage. Since the current causes losses in the conductors, also the efficiency of the inverter varies. Another factor that has an impact on the efficiency is that some parts of the solar inverter, such as the control system, consume energy

independent of the output power. Hence, tracking the maximum power point is very important. Later in this thesis, a simulation test for tracking the maximum power point is introduced.

### 3.3 Automated testing process

The automation process of testing the control board software consists of several phases (Fig. 3.5). At first, the source code is compiled, meaning that the executable code is created from the source files by using a suitable software construction tool. After the code is successfully compiled, binary files are uploaded on the control. Once this is executed, the software is ready to be tested and a comprehensive set of tests is run for testing the functionality and reliability of the software. The results are then published as reports, which provide information of the final state of the build.



*Figure 3.5. Outline of the process.*

In this implementation, the system tests are executed on software level, meaning that no real voltage is fed to the system. Instead, the voltage references are fed to the voltage variables of the inverter artificially by computer. In this manner, the behavior and functionalities of the inverter can be simulated and tested. In practice, the solar panel would feed the voltage to the solar inverter in which case the voltage sensors provide information of the voltage levels to the control system of the inverter.

### 3.4 Test automation framework

Test automation framework is an infrastructure, which provides a complete solution where different tools work together providing a common platform for the software testers to utilize it. With a proper test automation framework, software testers can focus on testing the software product instead of worrying about developing the test environment. A good test automation framework is general enough to provide functions that help a tester develop automated tests for all the different components of the delivered software system. It helps automate the execution and result analysis of test cases. Test automation framework should also be easily extensible so it can evolve as

the software system evolves. The objective of a test automation framework is to facilitate the development of automated test solutions. [4]

### **3.4.1 Automated tester framework**

Automated tester framework (ATF) is a framework, which is developed at ABB Drives. Therefore, it was chosen to be used in the automated testing system created in this thesis. It is compatible with Visual Studio, which is a development software utilized in this thesis for programming tests. In other words, ATF is a template that can be used in Visual Studio environment.

ATF consists of Gallio, MbUnit, and TesterAPI library. Gallio is an open, neutral, and extensible system for .NET framework that provides tools, runtime services and a common object model that any number of test frameworks can leverage. Gallio is able to perform not only the system tests of this thesis, but also unit tests and integration tests. In addition, Gallio provides appropriate interfaces that are easy to utilize. MbUnit is a unit-testing framework for .NET, which provides a multiple of useful features that make testing more practical. [9]

Gallio provides a graphical user interface, Gallio Icarus (figure 3.6), which provides a practical insight to the utilization of Gallio making it is easy to realize how it functions. It gives detailed information about the state of the tests in real time. In creation of the automation system of the thesis, the command-line test runner of Gallio, Gallio Echo (Fig. 3.7), is deployed. It has a progress monitor that provides feedback on the status of a test run and a brief summarize of the result. In the end of the tests, Gallio provides an extensive report, which indicates all the essential information concerning the final status of the tests. Gallio can publish these tests in several different formats such as html, xml, and mhtml among others.

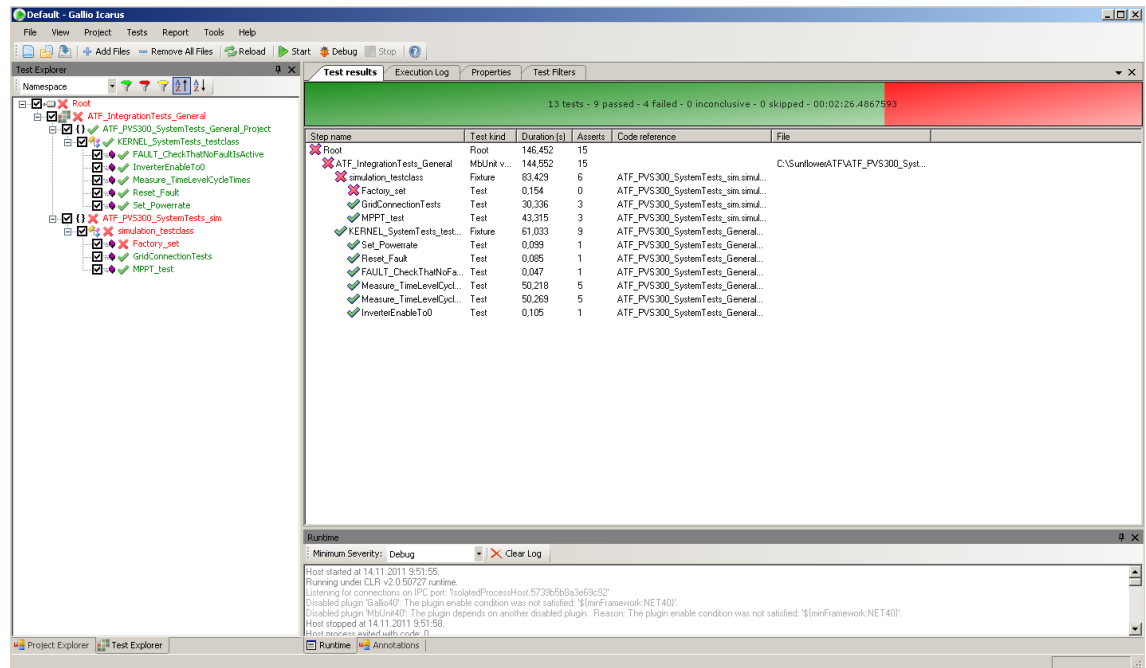


Figure 3.6. Gallio Icarus.

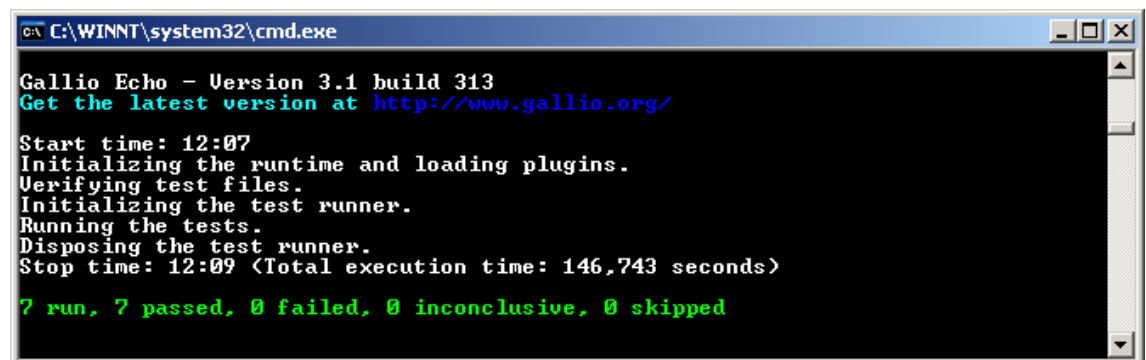
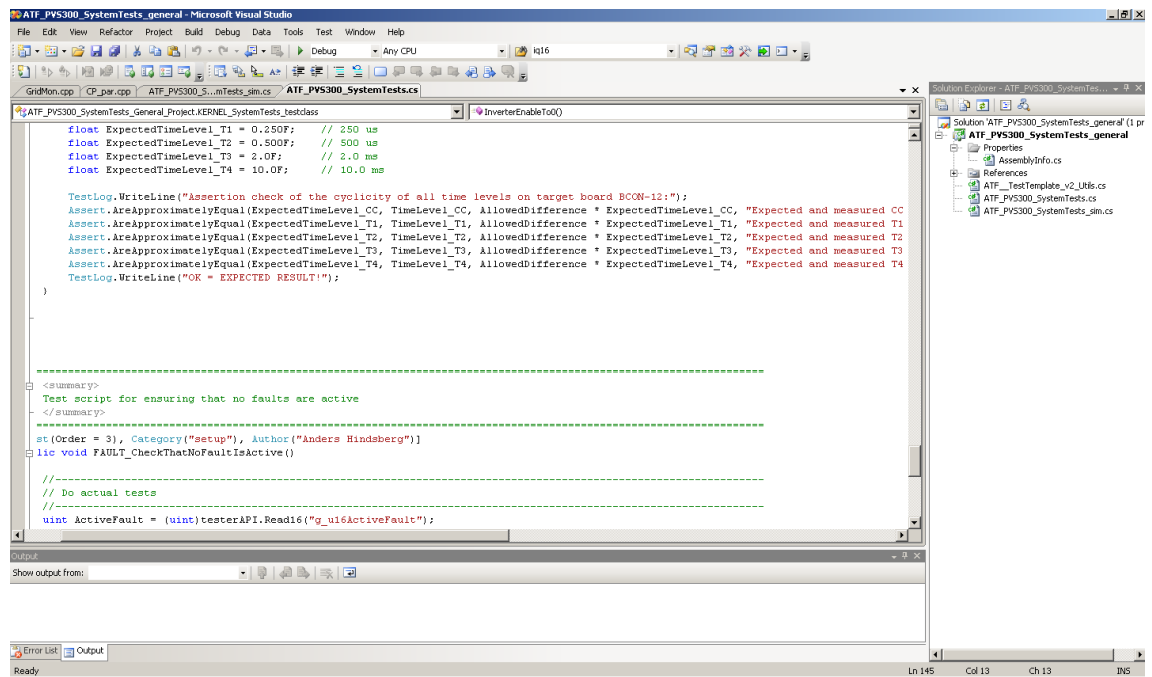


Figure 3.7. Gallio Echo.

### 3.4.2 Visual Studio

Visual Studio (Fig. 3.8) provides a set of templates, features and integrated development environment. It includes a few programming languages, such as C, C++, VB.NET, and C#, and it provides support for multiple programming languages via language services. It utilizes .NET Framework as the software framework. The basic libraries of .NET Framework provide support to, for example, user interface, data access, database connectivity, and network communications. Basically, it is intended to be utilized by applications on Windows platform. Software developers produce software by combining .NET Framework and other libraries with their own source code. The extensibility of Visual Studio allows writing own project templates and add-ins. [16; 17]



**Figure 3.8.** Visual Studio.



## 4 AGILE METHODS

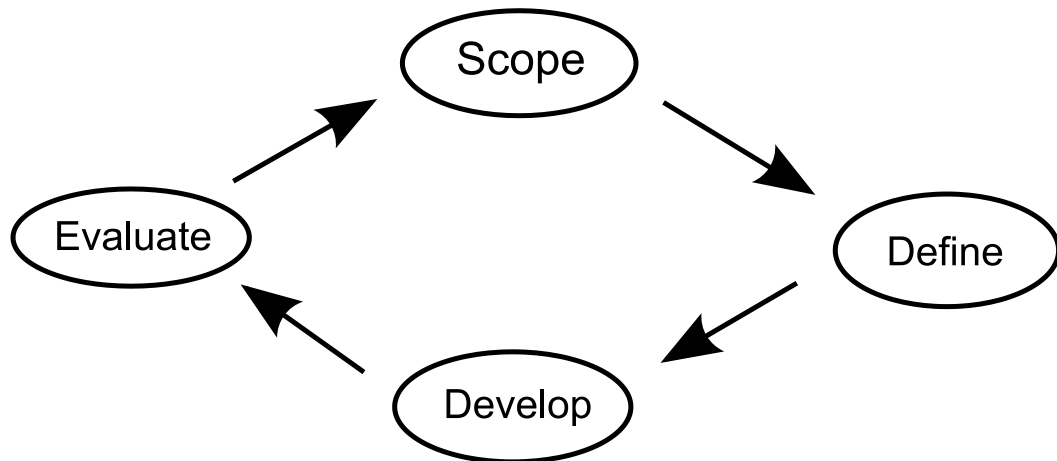
This chapter concentrates on presenting the main concepts that have been introduced into the automated testing system. The objective was to create an agile and modern system, which provides benefits to the development team. The main concentration is laid on introducing the concept of continuous integration, which is considered as a promising solution on the vision of the future in order to improve the efficiency of the development team. Subsequently, the concept of version control system is introduced in the end of the chapter.

Agile methods provide a method of programming based on iterative incremental development with each iteration taking few weeks of time and resulting in executable fraction of the complete product [19]. Agile methods are well suited for embedded systems development since they enable reducing of cycle times. They provide solutions for organizations involved in the lifecycle of embedded systems, which have to manage the growing operational and environmental complexity that these systems face. The more volatile the requirements, and the more experimental the technology, the more agile methods increase the odds of success. However, their utilization has not become a widespread practice. [25]

Finding the right tools and techniques to support test-driven development without compromising on the efficiencies already gained with homegrown tool suites is the challenge for organizations attempting to utilize agile methods. The development team is more effective when it can decrease the cost of moving information between people, and if the time elapsed between decision making and seeing the consequences of the decision is minimized. It is also claimed that agile development relies heavily on socialization through communication and collaboration to share and access tacit knowledge within the team. [25]

Technical issues, such as testing and requirements, as well as organizational issues, such as knowledge transfer, process tailoring, culture change and support infrastructure development, have to be prepared to effectively adopt agile methods in the subject of embedded systems development. Agile methods require the development of the right organizational infrastructure, including appropriate tools to support the software lifecycle, as well as creating an environment that supports collaboration and communication. This means taking advantage of the latest technology, which often leads to changing software requirements. One of the greatest challenges associated with agile adoption is integrating them with the existing environment. Agile methods represent a radically different fashion of performing the work associated with systems development. Basically, agile methods are about continuous way of developing the

product efficiently by using a testing method that provides frequent feedback of the state of the product. Figure 4.1 provides the idea of the loop that is performed repeatedly when working with agile methods. It means that possible sections of improvement are continuously searched. Therefore, constant development is continuously pursued. [25]



**Figure 4.1.** Agile methodology.

Agile methods enable delivering working and useful software rapidly. Speed is essential because delivering software is always cost-associated. Delivering software fast is also important because it allows verifying whether the new features and bug fixes are useful. The customer creates hypotheses about which features and bug fixes are useful to users. They remain hypotheses until they are in the hands of users who vote by choosing to utilize the product. Therefore, minimizing cycle time, so that an effective feedback loop can be created, is extremely important. [11]

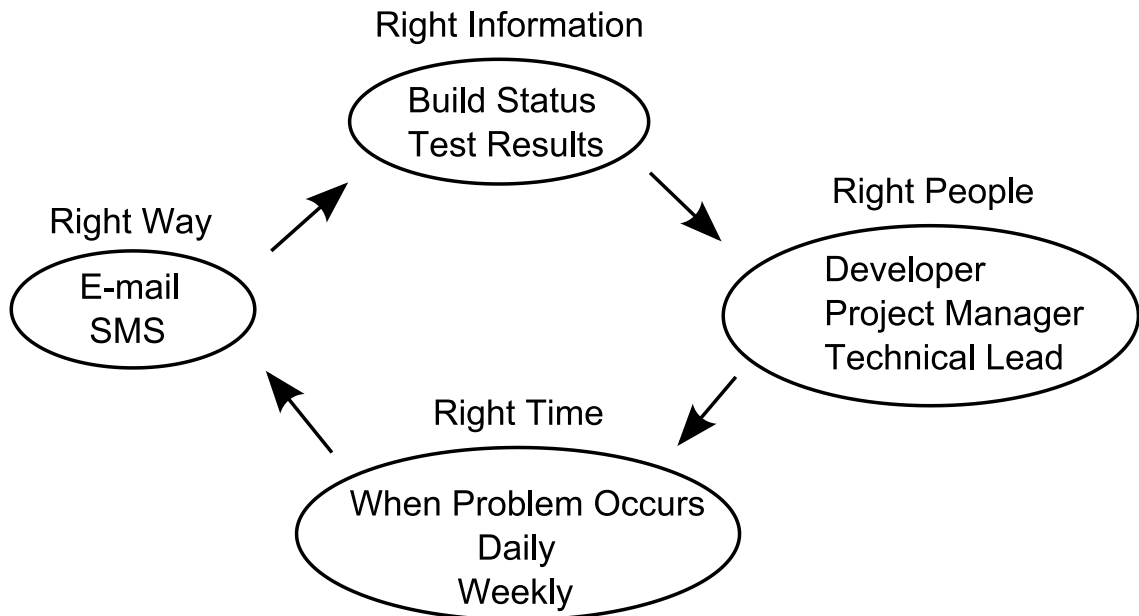
## 4.1 Continuous integration

Continuous integration offers a fast feedback providing method for software development. The reason for pursuing rapid feedback is in achieving an opportunity to find and fix problems fast throughout the development cycle. This helps in reducing assumptions on a project. Continuous integration is not literally continuous, but more likely continual. Frequency of the testing can be defined as wanted, for example, every five minutes or each time a new software update is available. In this manner, continuous integration enables automated tests constantly and repeatedly from the deployment until the end of the project. [7]

Developing software requires planning for change, continuously observing the result, and incrementally correcting the code based on the results. This is when continuous integration provides remarkable assistance. Immediate feedback is provided to the developer immediately when the build is executed. This method encourages

developers to make changes in the code, since even when the build breaks, immediate feedback is provided. Sending information of the latest build results to the right persons is vital. In addition, it has to reach them fast with correct information (Fig. 4.2).

[7]



**Figure 4.2.** Continuous feedback. [7]

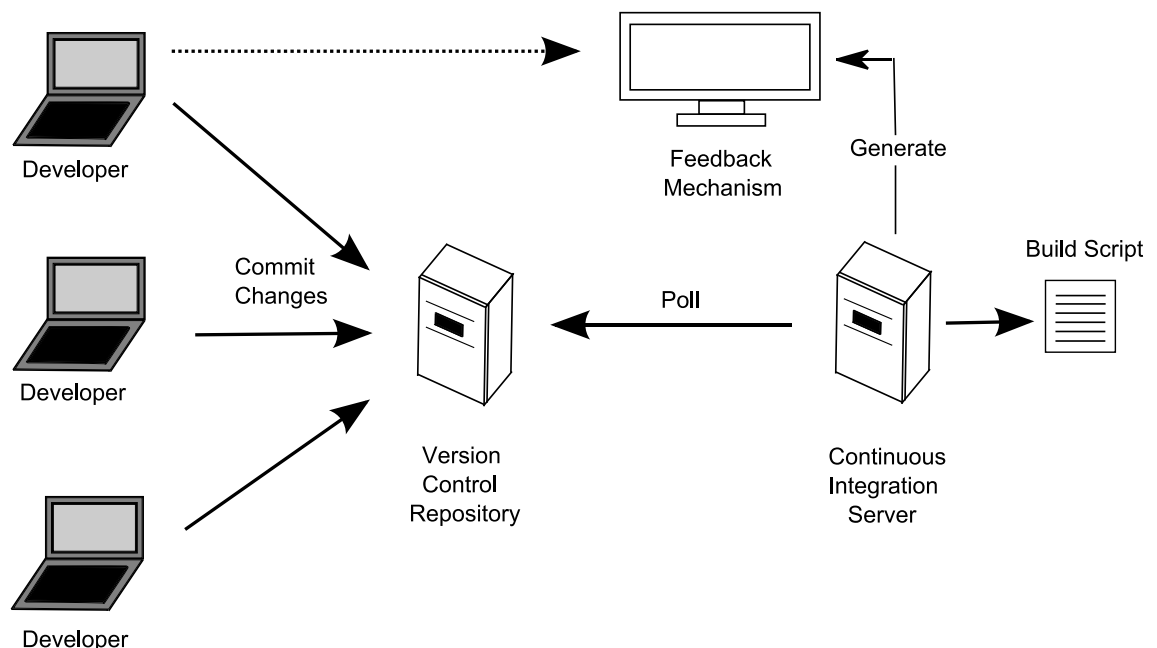
Most software developed by large teams spend a remarkable proportion of its development time in an unstable state. This is because the system and acceptance testing are not performed until the end the project. As a result, lengthy integration phases are scheduled at the end of development to allow the development team to get the branches merged and the application working. These integration periods can take a vast amount of time, which is often difficult to predict. In the worst case, teams can find that their software does not fit for purpose. [11]

The objective of continuous integration is that the software is always in a working state. When working most efficiently, continuous integration requires that each time a change is committed to the version control system, the entire application is built and an extensive set of automated tests are run to test its functionality. The development team should immediately fix any problem occurred if the build or test process fails. In this manner, the software is proven to work with the new changes when sufficiently comprehensive set of automated tests is run. By using continuous integration efficiently, software can be delivered faster and with fewer bugs than without it. With this practice, bugs are caught earlier in the delivery process when they are easier and cheaper to fix, providing cost and time savings. [11]

Certain prerequisites exist to start efficiently with continuous integration. First, a version control system is required for storing data. In the literature, all the files including code, tests, scripts and all the other files needed to create, run, install, and test the application, and basically everything that can change, should be stored in the

repository. The second prerequisite is to be able to run a build in an automated fashion from the continuous integration environment. A build may consist of the compilation and testing among other things. It acts as the process for putting source code together and verifying that the software works as cohesive unit. Thirdly, commitment from the developers is required, since continuous integration is a practice, not a tool. Every developer has to check in small incremental changes frequently to mainline and agree that fixing changes that break the application are the highest priority. Continuous integration will not lead to the desired improvement in quality if the developers do not adopt the commitment. [11; 7]

The basic idea of using continuous integration is simple. The integration tool needs to know where to find the source control repository, what script to run in order to compile, and run the automated commit tests for the application, and how to report if the latest changes cause tests to fail. The continuous integrations software polls the version control system for new possible commits. Commits can be made to source code, configuration files or other files included in the repository of the project. If new commits are found, the continuous integration tool checks out the latest version of the software, runs build script to compile the software, performs the tests, and notifies of the results in a way described in advance. In addition, it provides access to test reports. Figure 4.3 describes this process. In the implementation of this thesis, the version system repository and the continuous integration server are located on the same computer. [11]



**Figure 4.3.** *The components of continuous integration system. [7]*

In practice, the integration cycle consists of few repeatable phases. If the last build is running, a developer must wait until it is completed. If it fails, the whole team should work to make the build successful again, since getting broken code is forbidden. When

the build is successful, the developer updates his code in his development environment from the version control repository and makes his changes. If there were unit tests, they would be run on local development machine in this phase to verify that everything functions correctly. Often a personal build is programmed to execute this feature, but only for unit tests, since they are quick to run and they do not require the entire automation system. When the personal build completes successfully, the code can be committed into the version control repository. In system tests, code is committed directly to the repository. The continuous integration tool runs the build with the new code automatically after noticing new changes in the repository. If the build fails, the developer should start fixing the problem immediately and should continue until the build completes successfully. [11]

Frequent check-ins to mainline of the repository is the most important practice for continuous integration to work accordingly. Consequently, changes stay smaller and they are less likely to break the build. When a mistake is done, a recent functioning version of the software always exists. Also, it is important to run a comprehensive set of automated tests, which verify that the application is working properly. A passed build should provide confidence on the program, not only satisfaction of a successful run. [11]

In unit tests, the build process has to be kept short. Otherwise developers will check in less often due to the frustrating waiting time caused by queues. In unit tests, ten minutes should be maximum time for a complex test. If not, then the tests should be optimized. Normally time of one or two minutes is pursued. [11]

In system tests, build process durations are longer, since they require a fully installed system. In large development teams with multiple committers, running system tests with every commit could lead to lengthy queues and disorder. If the delay is too long, developers move to other activities and one of the primary benefits of continuous integration is not realized. Therefore, these types of tests are often run with periodic builds. As the software development team in the project, in which this thesis is written, consists of only few developers, the system tests are run with each commit. As more developers become related to the software development over the time, using periodic build should be considered. Especially, as the code base increases during the project, running all written tests can take an extremely long time for a build to complete. [11]

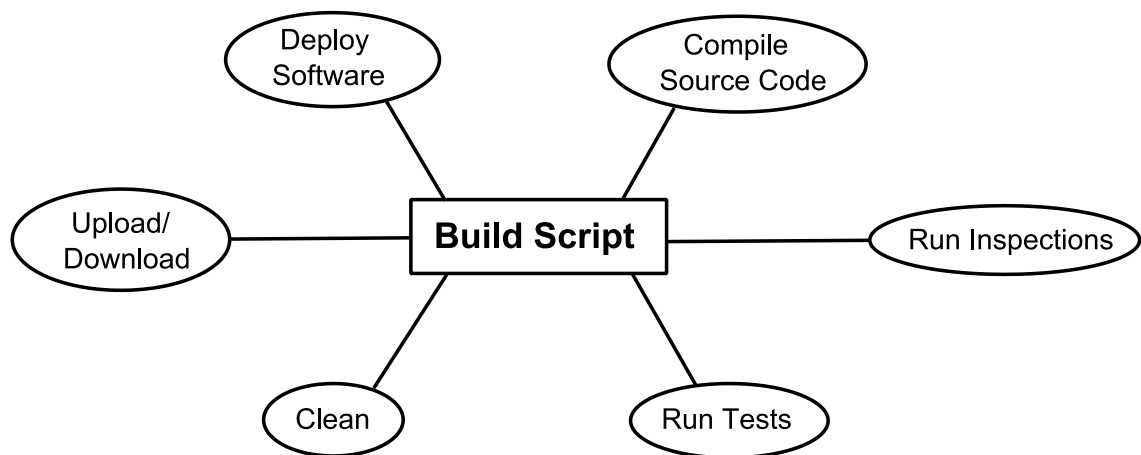
#### **4.1.1 The value of continuous integration**

There are several reasons for deploying automated software testing by using continuous integration on a project. First of all, it helps in reducing risks. Defects are detected and fixed early and health of the software is always known. Secondly, the use of continuous integration enables reducing repetitive processes. In this manner, time, costs, and effort can be saved. In addition, it can be assured that the process runs the same way every time. In this manner, human mistakes can be avoided as well. Thirdly, deployable software can be generated at any point in time. Furthermore, continuous integration

enables better visibility for the project, since a recent status is always available. Also trends in builds success or failure are possible to notice. Overall, by using continuous integration practices efficiently, improved confidence in the quality of the software is achieved since the impacts of the code modifications are indicated. [7]

#### 4.1.2 Build script

A continuous integration server requires a command script to execute tests in an automated manner. This is called a build script. A build script is a single or a set of scripts used for, for instance, compiling, testing, inspecting, and deploying software. They do not provide continuous integration themselves, but they can be used for automating the software build cycle. Build scripts define what is done and in which order in a build. They can be written in many programming languages. In this thesis, windows batch files were used as build tools since the programming language is well known and easy to identify. Fig. 4.4 provides examples of what can be performed via build scripts. [7]



**Figure 4.4.** Build script.

In this thesis, all the other tasks of the figure above are executed by the build scripts except of running inspections due to time constraints. Running inspections would be a useful task to include in the continuous integration process. In this manner, the style of the code, code duplication, code complexity can be inspected as well as code coverage can be assessed. Code coverage signifies the percentage of the code, which the written tests cover. This is extremely useful knowledge as tests are written to verify that the product functions properly. Deploying software means basically making the software available for use, for instance, releasing or installing and activating the software. [7]

### **4.1.3 Continuous integration server**

Continuous integration server is dedicated to execute the integration builds. It has all the necessary software components installed to perform the automated test cycle in created environment. Assumptions about environment and configuration can be declined, when a separate machine, a continuous integration server, is dedicated to integration builds. Typically, it is common that workstations have slightly different dependencies and configurations. This is when a dedicated integration build server provides an effective solution to the problem. Additionally, the fact that the computer is only used for integration tasks is a major advantage. The better the performance of the dedicated integration build computer is, the faster it can perform its tasks. [7]

### **4.1.4 Continuous integration tool**

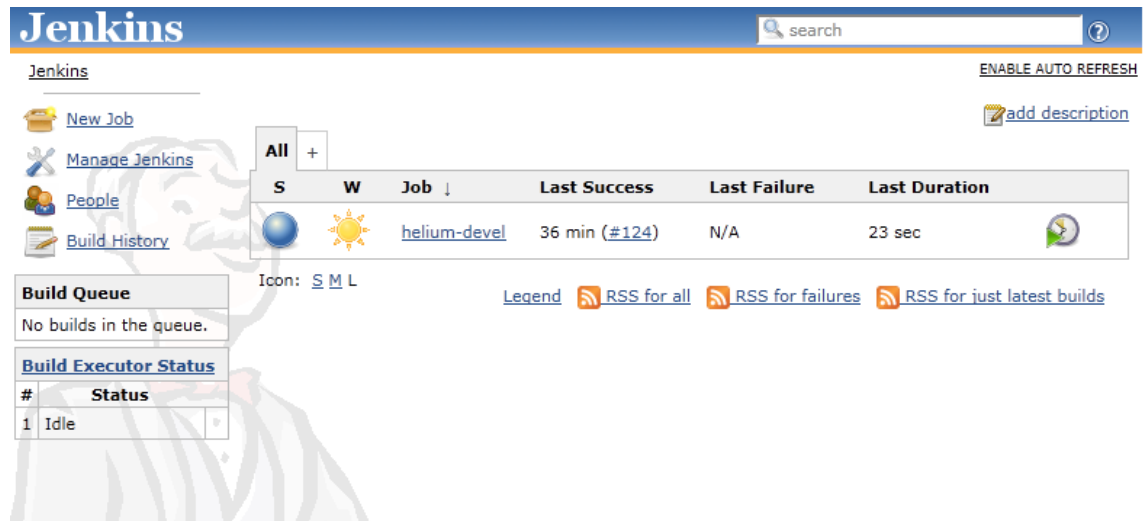
Continuous integration tool is running on the continuous integration server. Typically, a continuous integration tool is configured to check for changes in a version system repository every few minutes. When it detects changes, it retrieves the source files and runs a build script or scripts accordingly. Complete build jobs are then published on a dashboard. Continuous integration tool can also be scheduled to build on regular intervals, such as every two hours, but then it would not be according to the terms of continuous integration. In addition, it can be manually triggered from the dashboard. [7]

Continuous integration tools have several additional features. They provide different kinds of feedback mechanisms, for example, e-mail and SMS. Basically plenty of feedback mechanisms are available via plug-ins for keeping developers updated of the states of the builds. Continuous integration tools display also a history of previous builds. In addition, they offer support for multiple version control systems and build scripting tools via plug-ins. [7]

#### **4.1.4.1 Jenkins**

In the system designed in this thesis, an open source program called Jenkins is used as the continuous integration tool. After stating Jenkins as an easy-to-use program, and compatible program with the chosen version control system, it was chosen to be the integration tool utilized in this implementation.

Jenkins has a very simple interface (Fig. 4.5). The dashboard displays build jobs one on the other and provides practical information about their states. A blue sign indicates that the previous job was run successfully and a red sign indicates that the last job was failed. Jenkins utilizes a weather metaphor to provide an idea of the stability of the build. Essentially, the more the jobs fail, the worse the weather becomes. In addition, Jenkins indicates when the last successful and failed jobs were run, and what was the duration of the last job.



*Figure 4.5. Jenkins dashboard. [14]*

#### 4.1.5 Crucial practices

There are some crucial practices that are mandatory for continuous integration to work properly when there are several developers developing the same software. First practice is to not check in on a broken build. Developers are always responsible to fix the problem immediately when a problem occurs. Otherwise there is a chance that the failure is compounded with more problems. In that case, much more time is required to fix the problem. Additionally, developers might get used to see the build always broken, meaning that the build remains broken at all times. [11]

Second practice discusses on unit tests. As mentioned before, the practice is to run all commit tests locally before committing to the mainline. It is a way to ensure that the generated changes actually work. The local copy of the project is refreshed through updating it from the version control system. Then the local build should be initiated and the commit tests performed. When the build is successful, the developer is ready to commit the changes to the version control system on the continuous integration server. Two reasons for this exist. First, someone else might have checked in before the other one and the combination of these changes cause tests to fail. Secondly, an artifact is forgotten to update to the repository causing tests to fail. Checking out and committing test locally in advance can avoid these problems and breaking the build. Modern continuous integration tools provide this feature as, for example, personal build. By using personal build, continuous integration tool takes the local changes and runs a build on the continuous integration grid, instead of doing it by hands. If the build fails, continuous integration server notices the developer, and if the build passes, integration server checks in the changes automatically. This practice works best when using distributed version control system, since it allows the users to store commits locally without pushing them to the central server. [11]



Third practice is to commit code early and often in order to achieve the benefits of continuous integration. Committing after each small task is preferable. By doing small changes only, a lot of effort can be saved when errors occur. The benefits of the version control improve when committing regularly. It is impossible to safely refactor an application unless the changes are committed frequently to the mainline. In this manner, the changes stay small and manageable and they are also visible for the other developers. Naturally, committing and getting broken code should be avoided. [7; 11]

Basically, the developers who check in the changes are responsible for monitoring the progress of the build until it passes its commit tests. If it fails, they should fix the problem with another check-in or revert to the previous version in version control. Fixing the code is always the desirable solution. The name of the committer stays in the version control system, which makes developers deliberate their changes. For instance, it might give bad reputation to the committer if a broken build is left, for example, over a weekend, and some other developers would have to solve the problem. [11]

When creating new pieces of functionality or fixing a bug, the developers should first create a test that is an executable specification of the expected behavior of the code to be written. These tests verify the design of the application and operate as regression tests as well as documentation of the code. As mentioned before, this is called test-driven development. [11]

## **4.2 Version control**

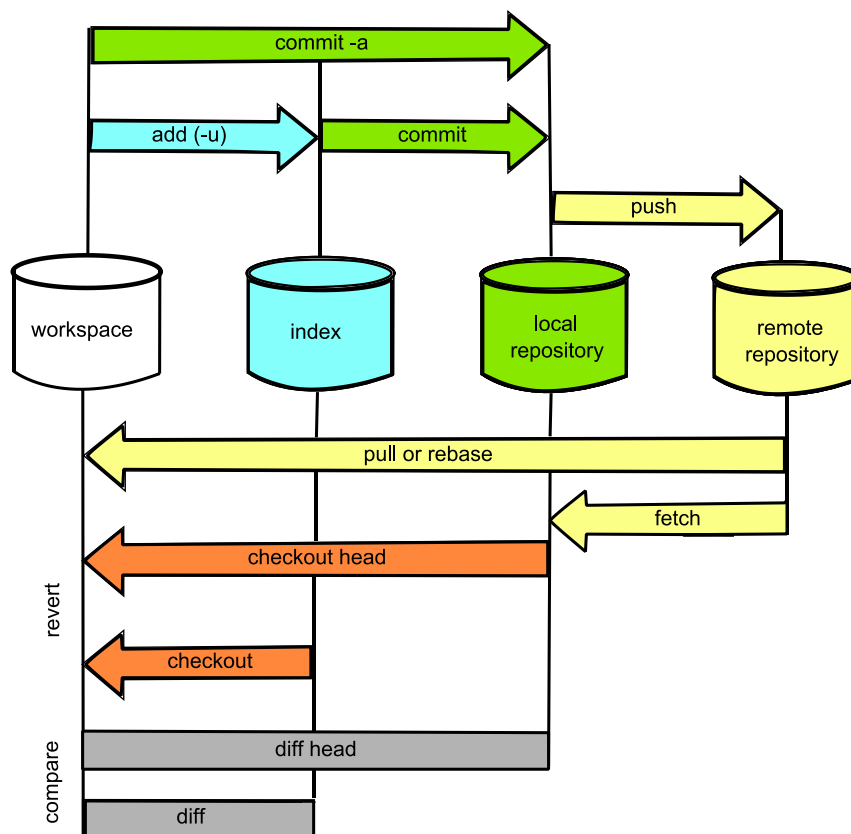
Version control system is a mechanism that enables keeping multiple versions of all the files of your software, so that after modifying a file the previous versions are still accessible. It gives developers the freedom to modify and delete, which is useful in creating new and getting rid of old ideas. Additionally, through version control people involved in software development are able to collaborate across space and time since they all can access to the version control system. [11]

Version control system maintains the complete history of every change made to the application. It indicates what was done when, by whom, and for what reason. The new changes have to be always commented when committing to the repository. These are valuable information when something goes wrong in the development process. Basically, not only the source code should be stored in the version control system, but also every single artifact related to the creation of the software should be stored there, so that a new member of the software development team can start working from scratch. In addition to source code, this means that also tests, database scripts, documentation, build and deployment scripts, libraries and configuration files for the application, compiler and collection of tools and so on, should be under version control. The objective is to store everything that can possibly change at any point of the life cycle of the project in controlled manner. [11]

As the aim of the thesis is to automate software testing process and thereby save time and enhance the quality, everything is depended on having a well-operating version control repository. When changes are checked into version control, they instantly become public and available to everybody on the team. Further, with continuous integration, it is possible to automatically execute testing every time changes to the software are performed. [11]

#### 4.2.1 Git

Git is an open source distributed version control system. What separates distributed version control system from the other version control systems, is the fact that each user keeps a self-contained, first-class repository on their own computer, effectively, creating their own branch. This enables many characteristics, for instance; commits can be easily modified, reordered, or batched up locally before sending changes to anybody else; having multiple copies of master repositories; pushing updates to a selected group without forcing them to take them; pulling updates individually from other users. Changes to the local working copy must be checked in to the local repository of the developer before they can be pushed to other repositories, and updates from other repositories must be reconciled with the local repository of the developer before updating the working copy. Figure 4.6 presents some of the most important Git-commands graphically as it describes the basic construction of Git. [11]



**Figure 4.6.** Git data transport commands and the construction of Git. [26]

Continuous integration works exactly the same with distributed version control system as with centralized version control system. Still the central repository is used as the mainline for developing the application. Distributed version control, however, provides several other possible workflows when preferred. [11]

In distributed version control system, like Git, developers can publish their own version for others to experiment with, instead of having to submit their patches to the project owner for committing them back to the repository of the project. This will lead to more experimentation, faster delivery of features and bug fixes, and faster evolution of projects. This means that commit access is not a bottleneck to developers creating new functionalities or fixing bugs. Repositories can be accessed via Git protocol or they can be published on Internet without any special web server configuration. Git also comes with several tools for visualizing and navigating a non-linear development history. [11; 5]

## 5 DESIGN OF THE TESTING SYSTEM

This chapter discusses on the facts how the system was created. First, the set-up is introduced. Then further details of the parts of the system are presented. Also, the execution of running builds on Jenkins is explained. Finally, feedback and backup mechanisms of the system are declared.

### 5.1 Introduction

The designing of the system contains both hardware and software design. At first, different options for software design were researched. Only programs working on Windows platform were considered. From the beginning it was clear that solutions or parts of solutions, implemented within ABB, would be under closer investigation, since knowledge and experience concerning these solutions should be reasonably available. A few solutions were examined. Also some external program options were considered for the continuous integration tool. Shortly, conclusion resulted in trying to exploit the promising software environments utilized at ABB Drives, thereby making ABB Drives more cohesive. The settings and configurations were supposed to be created, as it would provide the best benefits to the working team in the enabled circumstances.

As the object was to verify, that all the features of the solar inverter software remain functional when new updates take place, the automated testing environment has to be designed accordingly. Especially time concerned questions, like how often tests have to be run, how much does it take time, and when should they be run, needed to be solved.

By implementing continuous integration into the automation testing system, the possibility to run tests whenever new changes of the code are made to the mainline is realized. In this manner, the functionality of the software can be always verified. Most importantly, the developers of the software receive feedback immediately about the state and the functionality of the software. Thereby, the developers can work accordingly. When a build is run unsuccessfully, repairing the software is of the highest importance in order to remake the software well functioning for the other developers to work on. Hence, continuous integration was considered as a useful tool for improving the development of the software.

A version control system is one of the priorities for a decent automation system when it comes to conserving the software. It provides safety to the development team by offering a way to reverse to a previous functioning version after noticing that unfavorable changes to the code have been made. However, the main objective is to fix

problems instead of reversing to the previous version. Even so, it provides more courage within the development team to commit changes to the code.

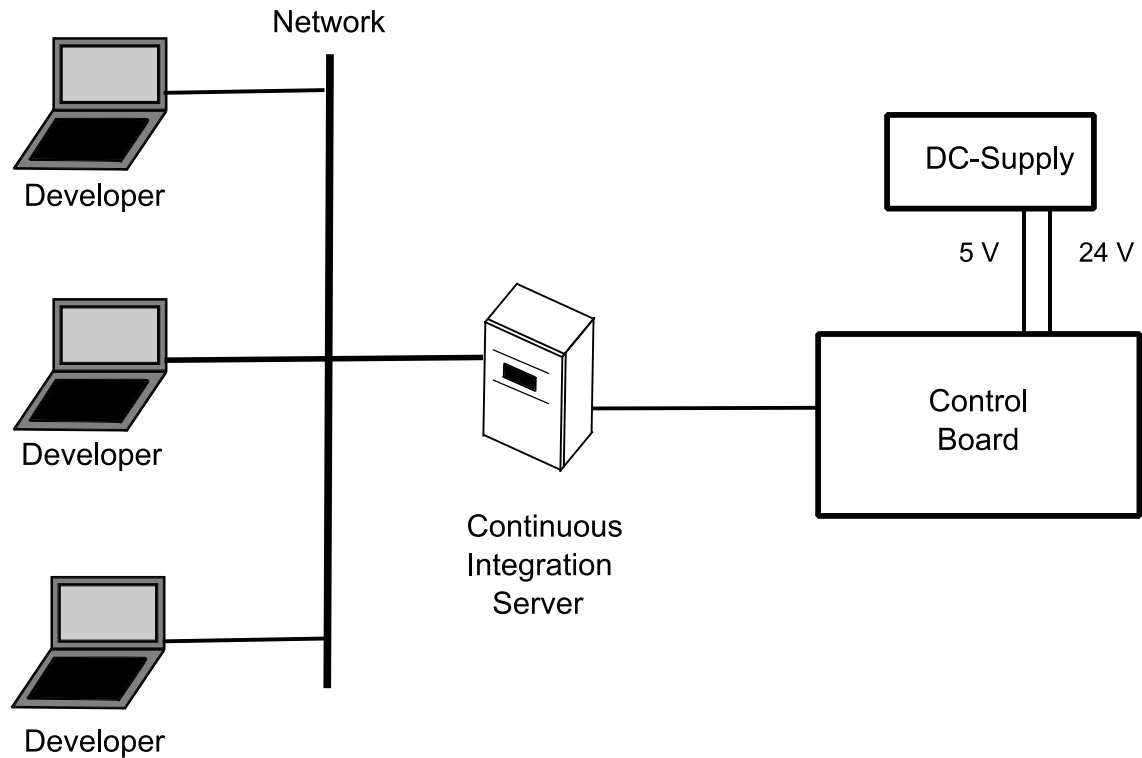
The core in automated testing system is naturally the test framework. It provides capabilities that enable a tester to create tests, execute tests, and analyze test results. The framework provides a collection of tests services that testers can use to develop automated tests. These services are software functions that can be called on for simplifying the common tasks that testers encounter when testing software.

In order to create an automated system for software testing, all the components have to work together properly. Continuous integration cannot be introduced if the continuous integration tool does not work together with the version control system. Additionally, the continuous integration tool has to be able to call on the test runner in order to execute the set of tests. Other factor that has to work properly is the connection between the control board and the continuous integration server as well as the connections between the developers and the continuous integration server.

The design of the testing system was started by using as simple tests as possible to verify that the complexity of the tests does not inhibit the testing of the automation system in the beginning of the implementation. Therefore, the best option was to commence by running only one functioning test at a time until the system is stated to operate correctly. Otherwise, the faults and defects in the tests would have complicated the testing of the system.

## **5.2 The set-up**

A simplified set-up of the automation system is illustrated in the Fig. 5.1. The developers commit their changes from their local workstations via network to the mainline of the repository, which is located on the continuous integration server. Whether the continuous integration tool is preset to poll for the changes in the repository or on periodical basis, it performs the specified tasks to the control board via USB-cable, which utilizes a proprietary communication protocol that is developed at ABB Drives. In the beginning, problems in functionality of the protocol were known to exist. Since then, software updates to solve the problem have been performed and improvements have been noticed.



**Figure 5.1.** *The set-up of the automation system.*

In order to function, the control board demands a power source with two different voltage levels. The communication section of the board is supplied with 24 volts and the processor section with 5 volts. The control board usually needs a load, for example the control panel, to awake after being shut down.

In the beginning of the thesis, the system was set up on a work desk. However, the objective was to find a decent location for it in a restricted space. The location should be close enough to the developers so that they could work with the system if any problem occurs. Additionally, the system is likely to be expanded in the future to cover not only simulation tests, but also tests using the operational set-up with real physical magnitudes.

### 5.3 Continuous integration server

A dedicated server enables a proper approach to introduce continuous integration in a software development team since it can concentrate only in running automated tests. A desktop computer was chosen for the purpose to prevent temptations of carrying it away from its location. In addition, it had a decent performance and it was quickly available. To ensure that all the programs are functioning on the computer, Windows platform was chosen to be installed on it.

Jenkins was installed on the continuous integration server as a Windows service (Fig. 5.2). In this manner, Jenkins starts always whenever the server reboots and can be

managed using the standard Windows administration tools. Hence, the continuous integration is always in operation whenever continuous integration server is on.

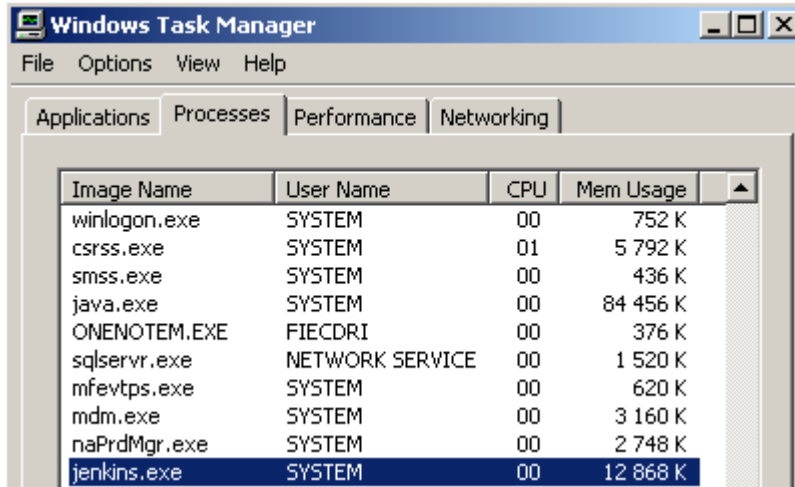


Image Name	User Name	CPU	Mem Usage
winlogon.exe	SYSTEM	00	752 K
csrss.exe	SYSTEM	01	5 792 K
smss.exe	SYSTEM	00	436 K
java.exe	SYSTEM	00	84 456 K
ONENOTEM.EXE	FIECDRI	00	376 K
sqlservr.exe	NETWORK SERVICE	00	1 520 K
mfevtps.exe	SYSTEM	00	620 K
mdm.exe	SYSTEM	00	3 160 K
naPrdMgr.exe	SYSTEM	00	2 748 K
jenkins.exe	SYSTEM	00	12 868 K

**Figure 5.2.** Jenkins installed as Windows service.

Version control system, Git, is installed on the continuous integration server for reserving data. In addition to the source code files, also test files, and batch files were included in the repository. In this manner, the continuous integration cycle is triggered also when test files or batch scripts are modified. This is practical, since also these files contain errors as likely as the source code files.

By using Git, a bare repository on the continuous integration server is created. A bare repository in Git contains only the version control information without any working files. This bare repository functions as the repository, from where the developers pull the changes of the code to their local ‘non-bare’ repository. In proportion, it also functions as the repository, to where the developers push their changes from their local repository. The ‘non-bare’ repository on the continuous integration server is then updated from the bare repository automatically, after which the code is ready for the compiling process.

In order to prepare the integration server for compiling the code, a proper environment has to be created on it. Basically, the environment requires the same tools that the software developers utilize for developing and checking the software. At first, a source code editor is required, which is utilized to create and modify the code of the software. Also, a construction tool is needed for generating final binary executables as well as a static code analysis software for analyzing the code. Additionally, a scripting tool is required to execute Python script files.

Next phase is to prepare the continuous integration server for uploading the code on the control board. The code is uploaded into the flash memory of the control board. Instead of attaching the control panel to its place, an USB-cable is connected to the control board with an adapter enabling the connection between the continuous integration server and the control board. To enable the data transfer, USB-driver has to be installed on the continuous integration server.

Two options exist for uploading the code to the control board. Both of them are realized by running a specified batch file, which calls certain tools included in the source code directory to install the software of the solar inverter on the control board. The first option is to upload a simulator software, that simulates the functions of the real operating situation. In this manner, the real measured variables can be replaced by artificial variables, which imitate the real behavior of a real operation of a solar panel. Additionally, natural variation can be eliminated, which enables pure software tests. In this implementation, this was the utilized procedure, since the system was performed in a simulated environment. The second option is to upload the real operational software of the solar inverter to the control board. This is the same software that is included in the commercial product which functions in real circumstances. In this software, the functioning of the product is not based on simulation, but on the real physical magnitudes.

In order to run system tests, ATF is installed on the continuous integration server. Basically, the idea is very simple; the test framework is needed for running tests and generating reports. Also, new tests can be created with the same environment. The test realizations are discussed in more detail in chapter six.

## 5.4 Local workstations

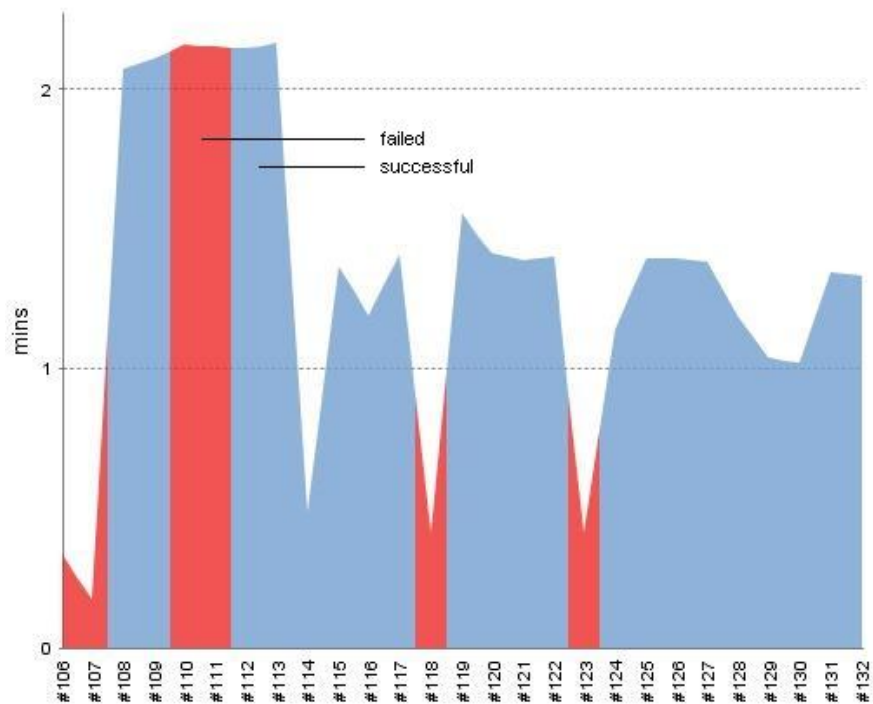
In general, developers have programs for developing and debugging the software. They may also have the control board with the power source and the connection for checking out certain issues. What they do not want to have are the plenty of performance and time requiring tasks of running the tests on their local computers. This is one reason among others why the devoted continuous integration server is implemented within the development team.

Since only the most novel code is suitable for development, the code is conserved in the git repository of the continuous integration server, where all the developers of the software have access. Hence, having Git installed on the local computer is a necessary requirement for all the developers. As a developer commences to develop the software, the first task is to clone the git repository from the continuous integration server to the local workstation to get the most recent software. The cloning of the repository on the local computer is only required once, since in the long term only the changes of the code are pulled from the continuous integration server. As all the developers have their local repositories, they have free hands to develop the software as long as they remember to push their changes back to the mainline of the repository on the continuous integration server for making it available for the other developers and, of course, for running the tests.

Developers can access Jenkins, which is run by the continuous integration server, from their own local workstation via a web browser. Naturally, the permissions have to be provided first. In this manner, they can easily view the state, history and other details



of the build from the Jenkins dashboard. Figure 5.3 presents the history of a Jenkins build job. Successful and failed build job are indicated with different colors.

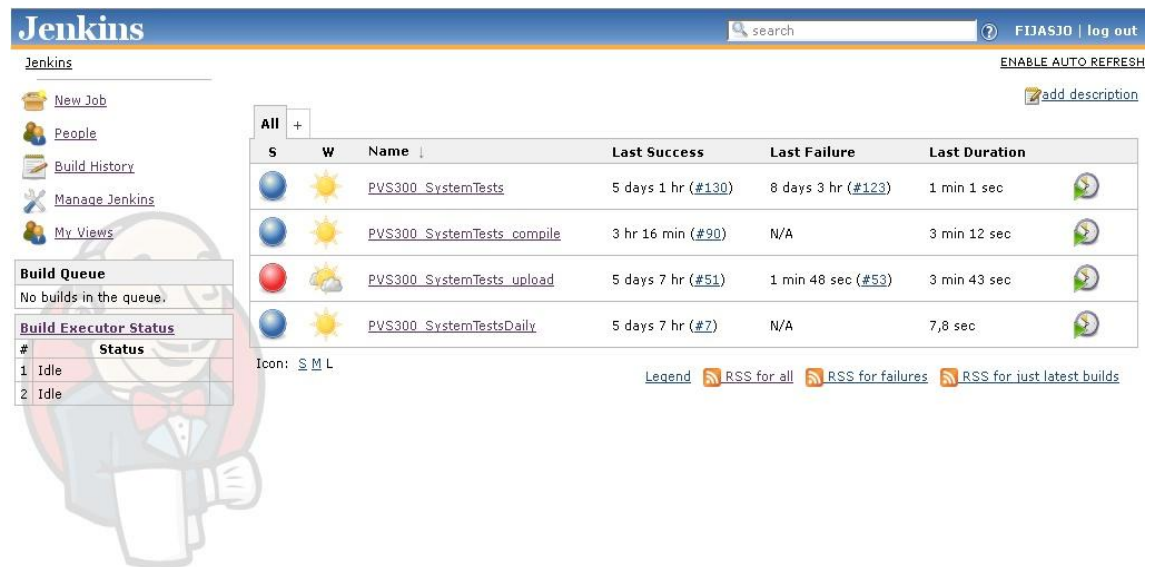


*Figure 5.3. History of a Jenkins build job. [26]*

## 5.5 Running the builds

Jenkins runs the build automatically when new changes are committed to the mainline of the repository located on the continuous integration server. It is scheduled to poll the repository every minute for changes. Three build jobs that form the build, are created to be run one after another. When Jenkins notices changes in the mainline of the repository, the first build job is triggered. The first build job executes a batch file, which compiles the source code and generates final binary files. Only if the compiling job is run successfully, the next build job for uploading is triggered automatically. It executes a batch file, which uploads the compiled software to the control board. The files that are uploaded to the control board can be specified by parameters in the batch file. For instance, the files intended only for the production version or the files intended only for uploading firmware version can be provided. When the build job for uploading is run successfully, the build job for running system tests is triggered automatically. This job executes a batch file that calls Gallio Echo test runner and provides it with parameters from the command line of Jenkins where the batch file is being called. These parameters are given when only certain test categories are intended to be run. Additionally, plenty of other parameters are given to Gallio Echo in the batch file. Closer introduction to the batch file is presented in Section 6.2.

The build process was divided into three job sections, since some advantages were noticed. At first, the failed job is easy to be recognized on the Jenkins dashboard since a red sign next to the failed job indicates it. Secondly, it improves the modularity of the process. In this manner, if the situation demands, only one job can be executed instead of the whole chain, since the jobs can be also triggered manually. The Jenkins dashboard including these build jobs is presented in Fig. 5.4.



**Figure 5.4.** Jenkins dashboard with the created build jobs.

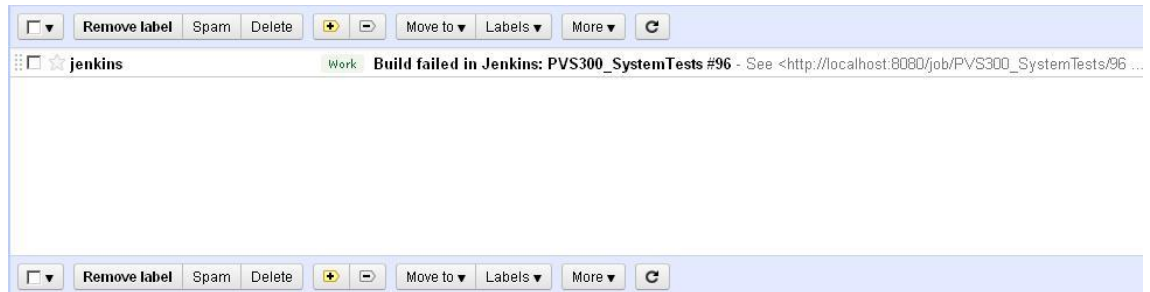
The daily build job is run every morning. At first, the job executes a batch file, which cleans the build-folder from the repository. This folder contains the generated binary files constructed by the former compiling process. The next compiling process will then regenerate the binary files. In this manner, the binary files are completely regenerated daily. The cleaning process is included only in the daily build but not in the continuous integration test process, since it is a time consuming process.

The daily build job runs the whole set of system tests by triggering the compile-job, which is the first job of the chain. In this manner, the whole chain will be run again. Especially, when a multitude of system tests exist and plenty of time is required to run the whole set of tests, the system tests are normally run only on a daily basis.

## 5.6 Feedback

The test process is considered as successful only if all the tree jobs are run successfully. In other words, all the jobs remain or become successful. This is when no feedback is sent via any feedback mechanism, since everything should be working as intended. All the jobs are configured to send an e-mail notification when they fail. Also, an e-mail notification is sent, when a job becomes successful after being failed. The e-mail addresses of the receivers are defined in advance. Also, separate e-mails can be sent to the developers who broke the build.

Various methods for providing feedback exist. The message can be provided via, for instance, text message, e-mail, or social network services. It is essential to reach the developer and provide information of the broken build. In this implementation, e-mail was the chosen method for delivering the message to the developers. The Fig. 5.5 presents a received e-mail from Jenkins after a failed build.



**Figure 5.5.** *A received e-mail from Jenkins due to a failed build.*

Gallio generates a report of every executed test. The report includes detailed information that helps developers repair possible defects. All the reports are saved on the hard disk of the continuous integrations server in Mhtml-format. The permission to this folder is shared to all the software developers so that the reports can be remotely analyzed. Figure 5.6 presents a test report generated by Gallio.



Figure 5.6. Gallio test report.

## 5.7 Backups

Backups are required to prevent loosing data when a critical fault occurs due to, for instance, broken hard disk or a computer virus. Hence, backups from the bare repository located on the continuous integration server are taken periodically. The process was realized by creating a Windows scheduled task, which runs a certain batch file. The batch file generates a zip-file of the bare repository and a script called in the batch file moves it to a location on a network drive reserving five backups: backups from last three days, a one-week-old backup, and a two-week-old backup. The script for conserving these files was introduced from another project. The backups are taken every night.

## 6 TEST REALISATIONS

In this chapter, the created tests and the ideas behind them are declared. The purpose was not to write automated tests for all the tests that were performed manually before. Yet, some tests were created to test some basic functionalities of the software and to modify some configurations of the product. Moreover, certain tests were created to simulate the basic operations of the product when the inverter is connected to the grid.

### 6.1 Introduction

In general, the black-box-method is utilized as the principle when writing automated system tests. Certain inputs are entered and the outputs are verified. Program 6.1 presents an example of a test created for verifying if resetting the ‘reset fault’ parameter also resets the ‘active fault’ parameter to a state where no fault is on.

```
//=====
    /// <summary>
    /// Test script for resetting fault.
    /// </summary>
//=====
[Test(Order = 2), Category("setup"), Author("Janne Sjögren")]
public void Reset_Fault()
{
    //-----
    // Check if the 'Reset Fault' bit resets the 'Active Fault'
    //-----
    byte group = 4;    // location for Active Fault
    byte index = 1;
    int fault = 0;
    testerAPI.ReadParameterValue(group, index, out fault);
    if (fault != 0)
    {
        group = 20;    // location for Reset Fault
        index = 9;
        TestLog.WriteLine("Fault was on in the beginning.");
        testerAPI.WriteParameterValue(group, index, 1);
    }
    group = 4;
    index = 1;
    int fault_finally = 0;
    // fault_finally can be 0 (fault is off) or 1 (fault is on)
    testerAPI.ReadParameterValue(group, index, out fault_finally);
    // successful if the Active Fault bit is 0
    Assert.AreEqual(0, fault_finally, "The fault is on." );
}
```

**Program 6.1.** A test script for resetting active fault.

ATF enables the use of multiple attributes that facilitate writing and running tests. To name some of those, as noticed in the program above, the test order can be determined for all the tests. Also, all the tests can be divided into categories. This is useful when instead of running the whole set of tests, only specified test categories are intended to run. The author-attribute becomes useful when tracking down details of a test, since the authors often know the ideas behind them.

ATF enables to write values directly to the parameters of the solar inverter software as well as reading from its parameters, which makes testing of the software possible. The locations of the parameters are the same than the ones found from the control panel of the solar inverter. The values are verified with the assert-command, which provides tens of different conditions for comparing values to the correct ones. In Program 6.1, the `fault_finally`-variable is inspected to be zero. If it is zero, the test returns a success. Otherwise, a failure is returned and a failure message: “The fault is on.” is printed. If any test returns a failure when running the set of tests, the Jenkins build returns a red sign indicating an unsuccessful build, ending up with a feedback e-mail sent to the specified developers. This is when the developers should realize, that something did not function as designed, and the code should be urgently improved.

## 6.2 Grid simulation tests

Two tests for verifying some of the most fundamental features were created by simulating the connection of the solar inverter to the grid. Connection to the grid, disconnection from the grid, and maximum power point tracking were considered as some of the most essential features of the solar inverter.

The connection to the grid and the disconnection from the grid were written in the same test case. As the intensity of the sunlight increases, the solar inverter connects to the grid when the DC-voltage exceeds a sufficient level, which is defined by the amplitude of the grid voltage. In this test, values are written directly to the variables of the solar inverter, which is enabled by the algorithms of ATF. The variable for the intensity of the sunlight is an `iq16`-number, which is a 32-bit fixed-point figure consisting of 16 integer bits and 16 fraction bits, created by the manufacturer of the processor of the control board. To ensure that the solar inverter connects to the grid, the simulated intensity is increased to  $800 \text{ W/m}^2$ . In order to write new values for the intensity, the values have to be converted to `iq16`-numbers. Shifting a 32-bit integer left by 16 executes that. Some delays are programmed in the simulation test, since the operation of the solar inverter has some delay. These delays were noticed from a separate simulator program. When the intensity has been increased, the connection is verified from a state-parameter. After the connection has been verified, the intensity is decreased back to  $100 \text{ W/m}^2$  and the disconnection is verified again from the state-

parameter. If the connection and disconnection have been confirmed as successful, the test returns success. Otherwise failure is returned. Program 6.2 presents this test.

```
//=====
    /// <summary>
    /// This test verifies that the inverter connects to the grid
    /// and disconnects from the grid.
    /// </summary>
//=====
[Test(Order = 1), Category("connection"), Author("Janne Sjögren")]
public void GridConnectionTest()
{
    //-----
    // Grid connections simulations and verifications
    //-----

    // set inverter Enable off
    testerAPI.Write32("u32_phndl_InverterEnable", 0);
    SleepMs(500);
    // Sunlight intensity 100 in the beginning
    testerAPI.Write32("iq16TestG", 100 << 16);
    SleepMs(500);
    // set InverterEnable on
    testerAPI.Write32("u32_phndl_InverterEnable", 1);
    SleepMs(500);
    uint CP_state = (uint)testerAPI.Read16("eCPSMState");
    Assert.AreEqual((uint) 2, CP_state, "CP state machine is not in
    Ready-state. Its value is {0}.", CP_state);
    // increase intensity
    testerAPI.Write32("iq16TestG", 300 << 16);
    SleepMs(1000);
    testerAPI.Write32("iq16TestG", 600 << 16);
    SleepMs(3000);
    testerAPI.Write32("iq16TestG", 800 << 16);
    SleepMs(8000);
    // verify that the inverter is connected to the grid
    CP_state = (uint)testerAPI.Read16("eCPSMState");
    Assert.AreEqual((uint) 6, CP_state, "CP state machine is not
    connected to the grid. Its value is {0}", CP_state);
    SleepMs(1000);
    // decrease intensity
    testerAPI.Write32("iq16TestG", 100 << 16);
    SleepMs(3000);
    // verify that the inverter disconnects from the grid
    CP_state = (uint)testerAPI.Read16("eCPSMState");
    Assert.AreEqual((uint)2, CP_state, "CP state machine is not
    disconnected from the grid. Its value is {0}", CP_state);
    // set inverter Enable off
    testerAPI.Write32("u32_phndl_InverterEnable", 0);
}
}
```

**Program 6.2.** A test script for grid connections.

Maximum power point tracking test commences by increasing the sunlight intensity to  $1000 \text{ W/m}^2$ . The maximum power point is defined to be reached when the intensity is  $1000 \text{ W/m}^2$  and the output current is nominal. This is indicated by the average power

correction variable, which should be greater than one when the desired maximum power point is reached. Since the maximum power point is not found immediately due to some delays caused by the tracking algorithm, the maximum power point is verified to be found every two seconds during one minute. The test return success if the maximum power point is reached. Otherwise failure is returned. The test for tracking the maximum power point is presented in program 6.3.

```
//=====
    /// <summary>
    /// This test verifies that the inverter sets to the maximum
    /// power point.
    /// </summary>
//=====
[Test(Order = 2), Category("connection"), Author("Janne Sjögren")]
public void MPPT_test()
{
    //-----
    // Grid connections simulations and verifications
    //-----
    // set inverter Enable off
    testerAPI.Write32("u32_phndl_InverterEnable", 0);
    SleepMs(500);
    // Sunlight intensity 100 in the beginning
    testerAPI.Write32("iq16TestG", 100 << 16);
    SleepMs(500);
    // set InverterEnable on
    testerAPI.Write32("u32_phndl_InverterEnable", 1);
    SleepMs(500);
    // intensity is increased
    testerAPI.Write32("iq16TestG", (Int32)(300 << 16));
    SleepMs(6000);
    testerAPI.Write32("iq16TestG", (Int32)(600 << 16));
    SleepMs(2000);
    testerAPI.Write32("iq16TestG", (Int32)(1000 << 16));
    // g_iq24PgAvgCycleCorr is used for checking if the MPPT is working
    // It should be > 1 when MPP is reached
    Int32 MPPT_check = testerAPI.Read32("g_iq24PgAvgCycleCorr");
    // MPP is the int value of the MPPT_check
    // It should be > 100% when MPP is reached
    int MPP = 0;
    // wait 2 seconds max 30 times to wait if the MPP is reached
    for (uint i = 1; i <= 30; i++)
    {
        SleepMs(2000);
        MPPT_check = testerAPI.Read32("g_iq24PgAvgCycleCorr");
        // g_iq24PgAvgCycleCorr should be > 1 when the MPP is reached
        if (MPPT_check > (Int32)(1 << 24))
        {
            MPP = (MPPT_check * 100) >> 24;
            TestLog.WriteLine("MPP reached {0}%", MPP);
            TestLog.WriteLine("Time needed for that was {0} seconds",
                (i*2+18));
            Assert.IsTrue(true);
            return;
        }
    }
    MPP = (MPPT_check * 100) >> 24;
}
```



```

TestLog.WriteLine("MPP reached {0}%", MPP);
// set inverter Enable off
testerAPI.Write32("u32_phndl_InverterEnable", 0);
// return false when the MPP (100%) was not reached
Assert.IsTrue(false, "Inverter did not reach the max power point in
about 1 min");
}

```

**Program 6.3.** *A test script for tracking the maximum power point.*

As well as the tests for grid connections, also the test for maximum power point required transforming iq-numbers to integers. Testlog-command enables writing notes of the progress of a test, which can be read in test reports.

### 6.3 Batch scripts

Batch files are executable files that contain text commands, which can be executed by the command line interpreter. In this implementation, batch files were utilized as build tools to be called by Jenkins. Each Jenkins build job has its own batch file which describes what is to be performed and in which order.

A test filter was created for specifying which tests are intended to be run. In this manner, the whole set of tests does not have to be run every time. The filter provides a reasonable solution in situations, where the whole set of tests would take hours to run and the development team would need faster feedback from certain tests.

As already seen in the test scripts introduced in this chapter, the possibility to categorize tests is provided. When the batch-file for running the system tests is called in Jenkins build, the intended test categories can be provided as parameters for the batch file. If no parameters are provided, the whole set of tests will be run. The created filter is then given to Gallio Echo as a parameter when Gallio Echo is called. Other parameters given to Gallio Echo are report type, report folder, report name format, show report, and the dll-file that includes the tests. Show report –parameter is given when the reports are wanted to be shown after running tests. The created batch script for running Gallio Echo with the filter is presented in Program 6.4.

```

@echo off
::-----
:: Batch file for running simple tests with Gallio Echo
:: Author: Janne Sjögren
:: Filter options:
:: Filter is needed to specify which tests will be run by giving the
:: specific categories as parameters. All tests will be run when no
:: parameters are given.
::-----
set reportType=mhtml
set reportFolder=C:\TestReports_SFCO_01
set file=C:\PVS300_SystemTests\bin\Debug\SystemTests.dll
:: filter * means running all the tests
set filter="*"
:: If command line parameters are given, go to one
if "%1" NEQ "" goto one
::If no command line parameters are given,use default filter,go to two
goto two
:one
set filter=Category:
set var=false
:again
if "%1" == "" goto two
if %var% == true (set filter=%filter%,%1) else (set filter=%filter%%1)
set var=true
shift
goto again
:two
set filterFormatted=%filter:=-_%
echo Filter used is %filter%
Gallio.echo %file% /sr /rt:%reportType% /rd:%reportFolder% /rnf:%
filterFormatted%-{0}-{1} /f:%filter%
echo Test report is saved in folder %reportFolder%
:end

```

**Program 6.4.** A batch script with the filter for running tests with Gallio Echo.

## 7 FUTURE PROSPECTS

This chapter discusses the development prospects of the created automated software testing system. Since the automated software testing system was created into the project starting from the very beginning, and the time reserved for the thesis was limited, plenty of ideas occurred were not yet implemented. However, the system will remain under constant development also after finishing this thesis.

Since version control system is a competent practice for conserving data, the contents of the repository located on the continuous integration server should be carefully considered. In this implementation, source code, binary files, test files and batch files were conserved in the repository. In addition, all the computer programs required to create the test environment, could be conserved in another repository. When new developers join the development team, the software would be easily available along with the source code and other data. This would also ensure that all the developers have the same versions of the programs, which is important when creating common testing environment for the developers.

As mentioned before, the objective of the thesis was to create an automated testing system for executing system tests since the project was already in the maintenance phase. In the future, automating also the unit tests should be considered when new projects are initiated because they are able to find problems early in the development cycle. Especially, since the unit tests are best to write before writing the actual code. Hence, also the use of test-driven development as the software development process should be considered. At the same time, test-driven development would function as the design and documentation method. Unit tests are relatively fast tests compared to system tests since they only test functions or simple parts of the code. Therefore, they would provide an appropriate test cycle before executing system tests. Of course, also integration tests would offer benefits in finding bugs early in the development process.

The test set should be expanded to encompass code coverage. By executing code coverage tests, information on how extensively the software has been tested, can be achieved. Only by testing the whole software, not only some parts of it, a decent confidence on the functionality of the software can be reached. Some code coverage software for .NET framework exists, but the subject was left outside this thesis due to time constraints. In addition to code coverage, also appropriate coding style verification could be included in the testing cycle.

Reviewing the code could very likely decrease errors in the code. The other software developers of the team should perform this before new source code is pushed to the main repository on continuous integration server. Reviewing would not only

function as an effective practice for identifying defects in the code, but also as a practice for recommending improvements in the realization of the code. However, further research is needed to generate a functioning practice for implementing reviews.

There were no professional testers in the development team during the execution of this thesis. As testing is a very important task in the development process, education for the testing people should be considered. Furthermore, a respondent for the testing system and for its development should be contemplated.

The hardware of the testing system is planned to be expanded in the future. Thus, functioning of the solar inverter will not only be tested by programming artificial simulations with computer, but by using physical magnitudes. This means that the system will cover the entire solar inverter. Therefore, operational tests can be programmed also by using real voltage magnitudes executed by power sources and grid simulators. This will also make the decision on the location of the testing system more important, since safety will have to be reconsidered.

Although this thesis was introduced when the project was already in the development phase, the created automation system will remain in the development team also when new projects are initiated. This will provide new opportunities for the development of the testing system, as new features that improve the verification of the functionality of the product, are valuable when the testing system develops along with the software.

## 8 SUMMARY

The purpose of this thesis was to design and implement an automated testing system for solar inverter software in order to make software testing more effective and to release testers from tedious software testing tasks. The idea was to investigate modern solutions to create a highly potential testing system for running tests automatically. The system was intended to create so that it will be easily upgraded in the future as well.

Options for different parts of the system were searched from literature and Internet. The resulting system consisted of three fundamental parts: test automation framework, continuous integration tool and version control system, each of them playing an important role in the system. Test automation framework provides an environment in which the tests can be programmed and executed. The continuous integration tool provides the possibility to perform certain tasks in specific order continuously or in periodical basis as it also enables sending immediate feedback of the test results. The version control system allows having back-ups of all the important files that are conserved in it. Through version control the software developers are able to collaborate across space and time since they all can access to the version control system. The utilization of version control system also enables the integration tool to run the tests every time a change is committed to the mainline of the repository.

As the automated testing system was created, some fundamental tests for verifying the correct function of the solar inverter were programmed. These tests were related to the grid connection and disconnection occasions. They were created to ensure that the most important functions are not compromised as new software updates occur.

The created automated testing system was taken into operation. It provides fluent running of tests as it provides feedback of the test results to the software developers. Based on feedback from the development team, it provides an environment for running tests automatically in competent manner thus making the software development more effective. Furthermore, the created system enables having always the latest up-to-date software under version control ready to be tested.

The system can be developed into many directions in the future. Many features for ensuring good software quality can be adapted. Also, not only performing system level tests, but also unit and integration tests will be considered. The possibilities provided by automation will bring a lot of savings in time and in resources. Hence, the possibilities will remain under close investigation. However, development of the testing system as well as improving the set of tests more comprehensive will continue after finishing this thesis. Especially, involving physical magnitudes to the testing system will take place in near future.

## SOURCES

- [1] Asikainen, A. 2006. Software testing of frequency converter, Lappeenranta, Lappeenranta University of Technology. 95 p. (in Finnish)
- [2] Beizer, B., 1990. Software testing techniques. 2nd Edition. USA. International Thomson Computer Press. 550 p.
- [3] Broekman, B., Notenboom, E., P. 2003. Testing Embedded Software. 1. Edition, Great Britain. 348 p.
- [4] Cervantes, A. 2009. Exploring the use of a test automation framework. IEEE Aerospace conference 2009, California Institute of Technology, Pasadena, CA, Jet Propulsion Lab. 1-9 p.
- [5] Chacon, S. 2011. Git: The fast version control system. [WWW].[Cited 30/6/2011]. Available at: <http://git-scm.com/about>
- [6] Dustin, E., Garrett, T., Gauf, B. 2009. Implementing automated software testing: How to save time and lower costs while raising quality, Boston, Pearson Education, Inc. 340 p.
- [7] Duvall, P.M., Matyas, S., Glover, A. 2010. Continuous Integration: Improving software quality and reducing risk, Boston, Pearson Education. 283 p.
- [8] End to end testing: Regression testing. [WWW]. [Cited 13/6/2011]. Available at: <http://www.endtoendtesting.com/index.php?page=regression-testing>
- [9] Gallio. 2010. [WWW]. [Cited 2/9/2011] Available at: <http://www.gallio.org/>
- [10] Heath, S. 2003. Embedded system design. 2. Edition, Burlington, MA, Newnes. 430 p.
- [11] Humble, J., Farley, D. 2010. Continuous Delivery: reliable software releases through build, test, and deployment automation, Boston, MA, Pearson Education, 463 p.
- [12] Janzen, D.S., Saiedian, H. 2006. On the influence of test-driven development on software design, IEEE Software engineering education and training, Kansas University, Lawrence, KS. 141-148 p.

- [13] Koopman, P. 1996. Embedded system design issues. [WWW]. [Cited 25/7/2011]. Available at: <http://www.ece.cmu.edu/~koopman/iccd96/iccd96.html>
- [14] Kwok, D. 2011. Derek Kwok's blog: blog about software development, techniques and discoveries; Jenkins and Github. [WWW]. [Cited 1/9/2011] Available at: <http://www.xairon.net/2011/03/jenkins-and-github-continuous-integration/>
- [15] Mallwitz, R., Engel, B. 2010. Solar power inverters, IEEE Integrated power electronics systems (CIPS), Niestetal, Germany, SMA Solar Technology AG. 1-7 p.
- [16] Microsoft: Visual Studio. 2011. [WWW]. [Cited 2/9/2011] Available at: <http://msdn.microsoft.com/en-us/vstudio/>
- [17] Microsoft: .NET Framework highlights. 2011. [WWW]. [Cited 2/9/2011] Available at: <http://msdn.microsoft.com/en-us/netframework/>
- [18] National Instruments: Maximum power point tracking. 2011. [WWW]. [Cited 2/9/2011] Available at: <http://zone.ni.com/devzone/cda/tut/p/id/8106> 1.9.2011
- [19] Sangwan, R.S., Laplante, P.A. 2006. Test-driven development in large projects. IEEE IT Professional, University Park, PA, Pennsylvania State University, Great Valley Sch. Of Graduate Professional Studies, 25-29 p.
- [20] Satalkar, B. 2011. Buzzle.com: Software testing techniques. [WWW]. [Cited 8/9/2011]. Available at: <http://www.buzzle.com/articles/software-testing-techniques.html>
- [21] Siniaalto M., Abrahamsson, P. 2007. A comparative case study on the impact of test-driven development on program design and test coverage, IEEE Empirical software engineering and measurement, Oulu, VTT Tech. Res. Centre of Finland. 275-284 p.
- [22] Software testing fundamentals: Unit testing. 2011. [WWW]. [Cited 6/6/2011]. Available at: <http://softwaretestingfundamentals.com/unit-testing/>
- [23] Software testing fundamentals: Integration testing. 2011. [WWW]. [Cited 6/6/2011]. Available at: <http://softwaretestingfundamentals.com/integration-testing/>

- [24] Software testing fundamentals: System testing. 2011. [WWW].[Cited 6/6/2011]. Available at: <http://softwaretestingfundamentals.com/system-testing/>
- [25] Srinivasan, J., Dobrin, R., Lundquist, K. 2009. 'State of the art' in using agile methods for embedded system development, IEEE Computer software and application conference, Vasteras, Sweden, Malardalen University. 522-527 p.
- [26] Steele, O. 2011. Oliver Steele: Languages of the real and artificial, Commit policies. [WWW].[Cited 29/7/2011]. Available at: <http://osteele.com/archives/2008/05/commit-policies>
- [27] Target, The-software-experts: Software process modules. [WWW]. [Cited 20/6/2011]. Available at: [http://www.the-software-experts.de/e\\_dta-sw-process.htm](http://www.the-software-experts.de/e_dta-sw-process.htm)