

LAPPEENRANTA UNIVERSITY OF TECHNOLOGY

Faculty of Energy Technology

Master's Degree in Industrial Electronics

Ekaterina Komova

AUTOMATED SOFTWARE TESTING IN MACHINE AUTOMATION

Examiners: Professor Olli Pyrhönen

Associate Professor Tuomo Lindh

Supervisors: Professor Olli Pyrhönen

D.Sc. (Tech.) Mikko Heikkilä

ABSTRACT

Lappeenranta University of Technology
Faculty of Energy Technology
Master's Degree in Industrial Electronics

Ekaterina Komova

Automated Software Testing in Machine Automation

Master's Thesis

2011

85 pages, 27 figures, 5 tables, 2 appendices

Examiners: Professor Olli Pyrhönen
Associate Professor Tuomo Lindh

Keywords: automated software testing, system testing, test cases generator, automated test framework

The problem of software (SW) defaults is becoming more and more topical because of increasing amount of the SW and its complication. The majority of these defaults are founded during the test part that consumes about 40-50% of the development efforts. Test automation allows reducing the cost of this process and increasing testing effectiveness. In the middle of 1980 the first tools for automated testing appeared and the automated process was implemented in different kinds of SW testing. In short time, it became obviously, automated testing can cause many problems such as increasing product cost, decreasing reliability and even project fail. This thesis describes automated testing process, its concept, lists main problems, and gives an algorithm for automated test tools selection. Also this work presents an overview of the main automated test tools for embedded systems.

ACKNOWLEDGEMENTS

The work was carried out at Konecranes, Hyvinkää during spring and summer of 2011.

I would like to express my sincere appreciation to the people who made this work possible.

First of all, I would like to thank my supervisor D.Sc. (Tech.) Mikko Heikkilä, who helped me with my thesis and who was always willing to give comments to my ideas.

I wish to express my thanks to Ari Lehtinen and Arto Engbom for giving me a great opportunity to write my thesis at Konecranes.

I want to thank my supervisors Professor Olli Pyrhonen and Associate Professor Tuomo Lindh for the possibility to work under their leadership.

I want to thank Professor Aleksandr Andryushin from MPEI for his contributions to this work.

Special thanks to Julia Vauterin who has made my live and study in Lappeenranta possible.

I am indebted to my friends for their support, friendship and new ideas.

I am grateful to my parents for their love and support

Hyvinkää, September 2011

CONTENT

1	INTRODUCTION	9
1.1	Background	9
1.2	Goals and delimitations.....	10
1.3	Structure of the thesis.....	11
2	TESTING.....	12
2.1	History of testing.....	12
2.2	Testing phases	13
2.3	Testing criteria	14
2.4	Test coverage metrics.....	15
2.5	Testing methods	17
2.6	Testing levels	18
2.6.1	Unit testing.....	18
2.6.2	Integration testing	20
2.6.3	System testing	21
2.6.4	Acceptance testing	21
2.6.5	Regression testing	22
2.7	SW development models.....	23
2.7.1	The waterfall model.....	23
2.7.2	The V model	25
2.7.3	The agile model	26
2.8	Conclusion.....	27
3	TESTING AUTOMATION.....	28
3.1	Manual and automated tests	29
3.2	Test automation concept	31
3.3	Automation testing metrics	32

3.3.1	Percent automatable	33
3.3.2	Automation progress	34
3.3.3	Test progress	34
3.3.4	Percent of automated testing test coverage	35
3.3.5	Defect trend analysis	35
3.3.6	Defect removal efficiency	36
3.4	Main test automation problem	37
3.5	Conclusion	39
4	CRANES	41
4.1	Crane. Component parts. Functions	41
4.2	Crane control system	42
4.3	The PLC based crane control system	43
4.4	Crane software architecture	48
4.5	Conclusion	49
5	AUTOMATED TESTING TOOLS	53
5.1	Tools selection algorithm	53
5.2	Test generators	55
5.2.1	Initial requirements	57
5.2.2	Investigate options	57
5.2.3	Refine requirements	59
5.2.4	Narrow the list	59
5.2.5	Evaluate the finalists	61
5.2.6	Conclusion	66
5.3	Test framework	67
5.3.1	Initial requirements	68
5.3.2	Investigate options	68
5.3.3	Refine requirements	69

5.3.4	Narrow the list	69
5.3.5	Evaluate the finalists	70
5.3.6	Conclusion	73
5.4	Conclusion.....	74
6	DISCUSSION AND CONCLUSION	75
6.1	Results of the work.....	75
6.2	Future work	77
	REFERENCES	79
	APPENDIX A. QML program of the crane power supply model.....	83
	APPENDIX B. Java model of the crane power supply	84

LIST OF FIGURES

Figure 1.1 Effect of automated testing. Adopted from (Juran 1999).....	9
Figure 2.1 The waterfall model of SW development models. Adopted from (Target)	24
Figure 2.2 The V model of SW development process. Adopted from (Melnik, Meszaros 2009)	25
Figure 3.1 Principle of manual testing process. Adopted from (Brown, Roggio & McCreary 1992).....	29
Figure 3.2 Automation testing. Adopted from (Brown, Roggio & McCreary 1992)	30
Figure 3.3 Parts of automated testing. Adopted from (Kanstrén 2010).....	31
Figure 4.1 The main crane component parts. Adopted from (Anonymous).....	42
Figure 4.2 The crane control system.....	43
Figure 4.3 The crane control system.....	44
Figure 4.4 Wire rope hoist	45
Figure 4.5 Crane SW architecture.....	48
Figure 5.1 Steps of the tools selection process	54
Figure 5.2 The MBT concept. Adopted from (Utting, Legear 2007)	56
Figure 5.3 Crane ON control logic	61
Figure 5.4 Model of Crane On logic (Conformiq, UML).....	62
Figure 5.5 Conformiq Designer Cover Editor	63
Figure 5.6 ModelJUnit Test Configuration window.....	63
Figure 5.7 Conformiq output viewing	64
Figure 5.8 ModelJUnit Result analysis: a) graph; b) results report	65
Figure 5.9 The ModelJUnit Main Window	65
Figure 5.10 Test execution workflow	67
Figure 5.11 Scheme of automated SW tests execution based on NI tools.....	71
Figure 5.12 Scheme of the real-time automated SW tests execution based on NI tools	71
Figure 5.13 HIL testing.....	72
Figure 5.14 Integrated testing	73
Figure 6.1 Automated SW testing.....	75
Figure 6.2 The automation process.....	77

LIST OF TABLES

Table 3.1 Manual and automated testing	39
Table 4.1 Main crane functions	46
Table 4.2 Standard IEC 61131 languages.....	50
Table 5.1 Results of tools evaluation.....	60
Table 5.2 Results of the test framework tools evaluation.....	70

LIST OF ABBREVIATION

AP	Automation Progress
CE	Consumer Electronics
CT	Cubicle Test
D	Number of Known Defects
DA	Number of Defects that are Founded after Delivery
DD	Defect Density
DRE	Defect Removal Efficiency
DT	Number of Defects Found during Testing
DTA	Defect Trend Analysis
EOT	Electrical Overhead Travelling
ESP	Estimated Stopping Position
FBD	Function Block Diagram
FD	Functional Description
FSM	Finite-State Machine
GUI	Graphical User Interface
HILS	Hardware-In-the-Loop Simulation
HTML	Hyper Text Markup Language
HW	Hardware
IL	Instruction List
IO	Input Output
IOLTS	Input/Output Labelled Transition System
LD	Ladder Diagram
MBT	Model Based Testing
NC	Numerical Control
NI	National Instrument
OS	Operating System
PA	Percent Automatable
PC	Personal Computer
PLC	Programmable Logic Controller
PTC	Percent of Automation Testing Coverage
QML	Qt Meta-Object Language
SCR	Screen
SFC	Sequential Function Chart
SILS	Software-In-the-Loop Simulation
ST	Structures Text
SUT	Software Under Testing
SW	SoftWare
T	Some Unit of Time
TCL	Tool Command Language
TGV	Test Generation with Verification technology

TTCN3	Testing and Test Control Notation version 3
UML	Unified Modeling Language
XML	eXtensible Markup Language
XP	Extreme Programming

GLOSSARY

Test case generator

A software tool that accepts as input source code, test criteria, specifications, or data structure definitions; uses these inputs to generate test input data; and, sometimes, determines expected results.

Test coverage

The degree to which a given test or set of tests addresses all specified requirements for a given system or component.

Test criteria

The criteria that a system or component must meet in order to pass a given test.

Test oracle

A source to determine expected results to compare with the actual result of the software under test.

Validation

The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

Verification

The process of evaluating a system or component to determine whether the products of a given development phase satisfy the condition imposed at the start of that phase.

1 INTRODUCTION

1.1 Background

The problem of software (SW) defaults is becoming more and more topical because of the increasing amount of the SW and its complication. The defects always cause material and other types of losses. Thus, companies spend much money to prevent and rectify defaults. Nevertheless, it is difficult to imagine the SW design process without defaults.

According to the experts, the cost of defaults increases to a considerable extent during design and product release. Rectifying defaults and design defects before final drafting release costs, for example, \$1. The cost is about \$10 after releasing, \$100 at the prototype stage, \$1000 at the pre-production stage and \$10,000 at the production stage. (Dhillon 1999) The cost of the defects elimination increases in two times after operation beginning, so it is important to expose the defects on the initial stage. Nevertheless, in accordance with the data, published by National Institute of Standards and Technology, the main amount of the defaults (about 70%) creeps in the project during the requirements phase and concept determining, but are discovered during testing (about 60%) and operation (21 %). (Dhillon 1999)

The majority of the defaults are founded during the test part. As usual, this part of the design is cumbersome, time-consuming and boring. It consumes about 40-50% of the development efforts and is a significant part of the design process. Test automation allows reducing the cost of this process and increasing testing effectiveness. The effect of automated testing implementation can be seen in Figure 1.1.

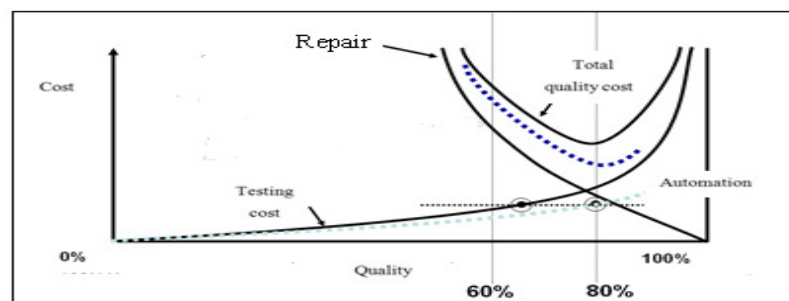


Figure 1.1 Effect of automated testing. Adopted from (Juran 1999)

As it is shown, companies' costs depend on the quality level. Quality cost depends on the costs of testing and defaults rectifying. The result cost has to be minimal. Using the test automation decreases the testing cost and, as a result, the minimum cost decreases too.

The test automation is a good tool to improve the product quality and decrease the product cost. Nevertheless, only small part of companies pays enough attention to this process and its improvement. According to the last research of 1000 companies that develop the SW 80% use no tools for automation testing and prefer to test the SW manually. About 80% of the left companies use only the simplest tools, 14% have different special tools and standard testing infrastructure. Other 5% implement testing services and organize special centers for experience and the project saving and changing. Only 1% of companies have total testing system for all projects. It can be explained by many problems that can be forced during automated testing implementation starting from automated testing tools choosing and ending with analysis of involved benefits. (Juran 1999.)

The test automation process consists of two parts: tools selection and implementation. Test automation tools selection is a project in its own right, and must be funded, resourced, and staffed adequately. (Fewster, Graham 2008) It represents an important process that should be done thoroughly enough; if not, it will cause many problems during the implementation phase. Unfortunately, this problem does not get proper attention.

1.2 Goals and delimitations

This master thesis concentrates on automated SW testing. It presents concept of SW testing and describes automated process. Also it lists main automated testing problems.

The work focuses on the automated SW testing in the crane area. So, it describes the crane control systems and main requirements for the automated testing. Also this work includes an algorithm for automated testing tools selection and overview of the main automated test cases generators and frameworks.

The testing tools implementation is beyond the scope of this work.

1.3 Structure of the thesis

This thesis consists of five large chapters.

Chapter 2 and 3 represent introduction to the work. Chapter 2 is a general introduction to the content of the thesis and includes the history of the SW testing and definition of testing criteria, phases. Also it describes test levels, methodologies and different SW development models. Chapter 3 focuses on the automated test concept, different test automation metrics and problems. In the end of this chapter the comparison between manual and automated testing is given.

Chapter 4 discusses modern EOT (electrical overhead travelling) cranes, its main parts, functions, SW crane architecture and configuration. Moreover, it introduces to the crane development and test process.

Chapter 5 comprises the tools choosing procedure, its requirements, and tools overview.

Chapter 6 presents the main results of this study and future possibilities in the application area.

2 TESTING

Before talking about automated testing, some words about testing itself should be said. The term “testing” appears in the beginning of the 19 century as a debugging synonym. Testing has traveled a long road since that time and became one of the most important quality criteria. Nowadays testing is not an act, but an intellectual discipline, which produces low-risks SW without much testing effort (Beizer 1990).

This chapter familiarizes with the basic principles and concepts of testing, describes different levels, techniques, and methodologies of testing.

2.1 History of testing

Testing is defined as:

The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component. (IEEE Std.610-12 1990).

The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate features of software items. (IEEE Std.610-12 1990)

An investigation conducted to provide stakeholders with information about the quality of the product or service under test. (Kaner 2006)

The definition of testing has been changing according to understanding of this process. In the beginning testing was allied to debugging. Testing was established as debugging or one of its parts. At the period debugging intended any necessary activity to expose the bugs. It was possible, when the programs were not complicated.

In 1957 Charles Baker revealed two different problems, which should be solved during programs writing: “make sure the program runs” and “make sure the program solves the problem” (Baker 1957). It became important to check if the program satisfied its requirements. The period was called debugging-oriented period. During that period the main idea of testing was checking all possible ways of the program execution and proving

that the program worked without fails. It should be observed that in many cases it takes much time because of the great number of inputs, their combination and paths, but it is still used in our days, for example, during acceptance tests.

In 1978 Myers in his “The Art of Software Testing” distinguished testing as “the process of execution a program with the intent of finding errors” (Meyers 2004). During the period, testing was a tool to show all possible program’s fails. In other words, this period is destruction-oriented.

In 1983 the Institute for Computer Sciences and Technology of the National Bureau of Standards published the report “Guideline for Lifecycle Validation, Verification and Testing of Computer Software”, which contained methodology, analysis, test activities and review to provide product evaluation during the SW life-cycle. In 1980x many changes have happened. First of all, the preventive tests, such as “Systematic Test and Evaluation Process”, became popular. Also tests planning, which became one of the most effective ways to prevent the defects, was involved. It was found that the testing process requires its own methodologies and tests should be involved in all design steps. Also the first tools for automated testing were created. (Adrion, Branstad & Cherniavsky 1982)

Since 1990, testing has included such ideas as planning, projection, supporting and executing testing. Since that time, testing became the main tool providing the product with the quality. Nowadays testing means a complex of different measures of analyzing the SW to establish its compliance with the customer’s requirements during the design process. (Gelperin, Hetzel 1988)

2.2 Testing phases

The testing process includes three phases. First of all, a test suite is created manually or automatically. Also the phase includes specifying both the desired and necessary properties of the test environment, any SW or supplies needed to support the test such as physical characteristics of the facilities including the hardware (HW), the communications and the system SW and others. Level of security that must be provided is determined as well.

Thereupon program running is performed on the test suite. Tests are executed on the determined input data set X and ordered output data set Y. As a result, a test log is created to provide a chronological record of the relevant details about the tests execution. For example, it comprises relationship between X and Y.

The third phase consists in testing results analysis and decision about testing continuation. (IEEE Std 829-1998 1998) The main part of this phase is special function Oracle, which detects, if outputs Y_o are conformed to the ordered outputs Y. The Oracle should have alternative way to find Y according to X. For example, even a programmer or a customer could realize the Oracle, calculating ordered outputs Y by their intuition. The Oracle displays, if there is some discrepancy between the ordered outputs and the real outputs, but gives no information about the way it has been computed.

The result of the third phase is decision if the SW passes the test or it should have some modifications and test repeating.

2.3 Testing criteria

It is impossible to test the program under all inputs and their combinations or by all paths, because testing should be ended and it is always limited by time and material resources. This problem does not have a solution in the general case and is the main problem of testing. Therefore, it is so important to choose a suit of tests, which will examine as many different situations as possible and does not exceed limits. Testing criteria are used to examine tracks, which have considerable differences, by systematic way. (Bourque, Dupuis 2004) A SW test adequacy criterion is a predicate that defines “*what properties of a program must be exercised to constitute a ‘thorough’ test, i.e., one whose successful execution implies no errors in a tested program.*” (Goodenough J.B. 1975)” The requirements to the ideal criteria were stated John Goodenough and Susan Gerhart:

1. criteria should be valid. It means criteria should show when the set of tests are enough to examine the program;
2. criteria should be reliable. Any two different test suits, which satisfy the criteria, should detect or not detect the same defects of the program;

3. criteria should be full. If program has an defect there should at least one test in the test suit, which could define the defect;
4. criteria should be easy to exam. (Goodenough J.B. 1975)

Unfortunately, there is no computable satisfied these requirements criterion that can be practically applicable.

There are several criteria types such as structural, functional, stochastic and mutational criteria. The criterion that is implemented depends on the type of the program, which is going to be executed. The structural criteria use information about the program. The functional criteria are formed during settling requirements to the program. The criteria of stochastic testing are formed to check special properties of the SW. Mutation criteria use Monte Carlo methods to check properties of program. Here are some examples of the criteria to understand the basic idea.

—*Statement coverage*. A statement coverage criterion requires execution of all statements in the SUT (software under test) at least once. The criteria type is widespread; testers are often generating test cases to execute every statement in the program. A test set, which satisfies the requirement, is weighed to be adequate according to the statement coverage criterion. Evaluation of testing adequately can be done according to the percentage of executed statements. The percentage of the statements exercised by testing is a measurement of the adequacy. (Hong Zhu, Hall & May 1997)

—*Branch coverage*. A branch coverage criterion requires exercise of all SW control transfers under testing. The percentage of the control transfers executed during testing is a measurement of test adequacy. (Hong Zhu, Hall & May 1997)

—*Path coverage*. A path coverage criterion requires execution of all paths from the program's entry to its exit during testing. (Hong Zhu, Hall & May 1997)

2.4 Test coverage metrics

It is useless to speak about improving testing, if there is no metrics to appraise the quality of this process. Testing consists of many parts and procedures, so it is impossible to

characterize it by one computing value. In the middle 1970x the term test coverage appeared to describe effectiveness of SW testing. (Beizer 1990)

Test coverage is the degree to which a given test or set of tests addresses all specified requirements for a given system or component. (Juran 1999)

The test coverage is not only the quality indicator of the tests. Also the test coverage helps to determine the program areas that are not tested by the existent test cases. So, new test cases are created based on the current test coverage. The main metrics of testing coverage are described in this section.

Testing ratio is one of the basic measures, which shows the percent of the executed SUT in relation to the all SUT. So, it can be found according to Eq.2.1.

$$Testing\ ratio = \frac{executed\ SUT}{SUT} \cdot 100\% \quad (2.1)$$

If the metric is, for example, 60%, it means that the tests cover 60% of the SUT code and do not 40%. Obviously, other tests that are under development should cover those 40% of the SUT.

Also *requirements coverage* is used to describe the testing process. As it can be seen from Eq. 2.2 this metric describes how many requirements that should be satisfied by the SW have been tested. This metric is useful during system and acceptance testing.

$$Requirements\ coverage = \frac{tested\ requirements}{specified\ requirements} \cdot 100\% \quad (2.2)$$

Architectural coverage is used during “white box” testing that proposes the SW structure and its sores code. This metric evaluates if every possible paths or other architecture units in each function have been followed. The architectural coverage can be found according to Eq. 2.3.

$$Architectural\ coverage = \frac{tested\ architectural\ features}{architectural\ features} \cdot 100\% \quad (2.3)$$

Code coverage shows how many statements of branches and paths have been tested (Eq. 2.4). This metric shows if every available statement is executed. The metric can be used without source code adaptation, but it is insensitive to some control structures. (Cornett 2011) So, this metric is used more by computational statements.

$$Code\ coverage = \frac{tested\ statements\ branches\ paths\ states}{statements\ branches\ paths\ states} \cdot 100\% \quad (2.4)$$

Other metrics can be created to describe the testing process in the specific way. All metrics are used in the different design steps and no one can be used alone to represent the current state of the design process, their combinations are used.

2.5 Testing methods

The SW is currently tested using different types of methodologies. The most widespread methods are methods of “black” and “white boxes”. Also there are such methodologies as assertion, mutation and “grey box”.

A "white box" methodology is performed by using information, which is internal to the tested SW (Meyers 2004). The SW testers normally use a white box test method when verifying SW's requirements (Coulter 2000). This kind of methods presupposes a source code or a specification of the program in the form of a control flow graph (Beizer 1990). Structural information is available for the developers of the subsystem and units. Therefore, this methodology is used for unit and integration testing.

"Black box" is one of the most important methods in SW testing. The SW is tested using a "Black box" method by interfacing to the SUT through its formal interface (Meyers 2004). Thus, the SW is considered as a "Black box" and there is no information about its internals, which is visible to the SW tester. This method exams, how the SW satisfies customer's requirements, and reproduces relationship between the SW and the environment. The documents, which contain customer's SW requirements such as SW Requirement specification and Functional specification, are cumbersome. Nevertheless, testing should be comprehensive and it makes “black box” method labor-consuming.

“Grey box” testing is a combination of all previous methods. The method is testing the SW from outside like the “black box” method. At the same time the tester has some structural information about the SW, which allows to choose correct test suit and to make the testing process more effective. “Grey box” testing validates that the SW meets its external specification and that all paths through the SW have been exercised and verified. (Coulter 2000)

Mutation testing is a one of the SW testing form, which is destructive in nature. (Coulter 2000) Mutation testing is actually used not for SW testing, but to test the adequacy of the SW tests. Mutation testing creates a variant of the SUT, which has some kind of defects. This mutant program is checking using the SW test cases. If test cases, which were created for the SUT, satisfy all requirements, then the mutant program should fail at least one of the tests. If a mutant and the original program produce different outputs on at least one test case, the fault is detected and the mutant is dead or killed. If the mutant program passes all test cases, it means that an insufficient number of test cases were provided to detect program operation. The percentage of the dead mutants compared to the mutants that are not equivalent to the original program is an adequacy measurement, called the *mutation score* or *mutation adequacy*. (Hong Zhu, Hall & May 1997)

2.6 Testing levels

The functionality of the modern systems is increasing constantly. (Winkler, Hametner & Biffel 2009) It causes, first of all, increasing added functionality that are implemented in the SW. Therefore, SW components become more complex. Systems requirements have been changed not only during the first step, but during all development process. The current automation systems have a code that constitutes interweaving of SW code and testing code. The code becomes hard to read and to modify during the development and the maintenance. Thus, systematic testing and efficient are hindered. The code requires complex testing during all development steps from the testing variants or the individual component to testing of whole system. Therefore, the testing process was discriminated between different levels such as unit, integration, system, acceptance and regression testing.

2.6.1 Unit testing

Unit testing is testing of different units, functions and classes of the program separately. The main purpose of this level of testing is an exposure of local defects of the algorithm in this init, and also the readiness the program for other development steps. As usual, unit

testing bases on "white box" methodology. Commonly, this level contemplates an environment around the unit that has stubs for test unit interface. They can be used to create inputs, analyze results and for other purposes. The units are widely used for revealing different logic and code algorithms defects.

Unit testing can use different principles. First, it can be based on the thread of execution. The criterion is, for example, call pair coverage, which means each program functions should be called at least once. Also unit testing can use the data flow. This principle permits to refer the undefined variables and prevent redundant assignments. It requires testing of all intercommunications that include reference to a variable and definition this variable.

The main problem of this level is deterministic of the test cases. The process has different steps (Prather R., Myers J. P., Jr. 1987). First of all, the flow graph is build. The variables, which should be tested, are detected using this graph. Then testing paths should be chosen. The process can be based on static, dynamic or release paths ways. The static way means that the test path is made longer and longer by adding new arcs until the flow graph output is reached. The method does not take consideration that the ways can be unrealistic. The dynamic way requires a system of tests, which covers at the same time different paths and data. This way allows taking account of realistic or unrealistic paths automatically. The main idea is adding to the previous path different parts so, that new way is realistic and covers the demanded elements. Paths way releasing means exposing real path from whole path set. The third step defines tests that release the chosen paths.

“Buddy testing” is a kind of unit testing, where tester and developer work together, forming a “buddy”. In this case, one programmer writes a code that is tested by another one and vice versa. It is a very effective and efficient testing practice. Each new function goes through the quick check, so test scripts are created essentially. Thus, quality is assured itself at this level. Number of the SW defects is decreased by the low step risk analysis. Therefore, the product quality is increased.

As conclusion, it can be said that unit testing is

- smallest testable piece of testing
- normally done by programmer
- done to expose local defects of algorithm in single init

2.6.2 *Integration testing*

Integration testing is testing of the system that consists of more than one unit. The main purpose of this testing is exposure the defects of interactions between different units. This testing level is developed unit testing, because it uses unit interfaces and requires an environment that has special stub instead of absent units. The main difference between the levels contains types of defects that are revealed by the levels. The defects determine methods of choosing inputs and analyzing. For example, the integration testing often uses interface cover methods such as analyzing of using different interface elements by calling functions, global resources, and communication tools.

Integration testing uses the "white box" methodology. The tester knows a structure of the program in detail right up to the units calling, which are included in the testing part of the SW. This level is used during unit assembly, which can be done by two different ways. If all units are gathered simultaneously, it is called monolithic. If there are some absent units, they are replaced by the test drivers that are developed in addition. The methodology requites many costs because of the complexity of the stubs, the adding test drives, the defects, but it allows parallelizing of the development especially in the beginning of the design process. If the integration testing set is increasing step by step by adding the units, it is called incremental. Adding the units can be done top-down or down-top. Top-down testing uses the stubs, the priority for the testing units, operations for external exchanging. If the way is used, such problems as developing of intellectual stubs, complex environment are faced. Also the parallel developing of the different units does not always bring to the effective realization of the units because the units, which have not been tested yet, are added to the units, which have been tested. Bottom-up testing uses different types of stubs too and often does not test concepts of the testing SW.

As conclusion, it can be said that integration testing:

- checks if interactions between units are correct
- models can be coded by different people
- includes monolithic testing
- includes incremental testing
- proposes bottom up (using drivers) and top down (using stubs)

2.6.3 System testing

System testing is exam a system in a general case and uses a user interface. It is very difficult and low effective to realize test path entire the program or exam the correct work of the single function. The main purpose of system testing is exposure of defects that are connected with whole program. For example, wrong using of system resources, incorrect functionality, inconvenient using and others.

System testing uses a "black box" methodology. The tester uses only inputs and outputs, which are available for the users, so the structure of the program is not the object. User's documentations are tested too. During the system testing level all required functions, stress condition, correctness of resource using and documentation, performance are checking. Because of the huge amount of the data the test automation is very effective during this level (Winkler, Hametner & Biffl 2009). So, system testing has more complex structure then the unit and integration testing systems.

As conclusion, it can be said that system testing:

- executes the whole system
- uses only inputs and outputs, which are available to the users
- is done not by one person, but by team
- cases are written according to high level design specification
- is done by automatic way
- uses simulates environment

2.6.4 Acceptance testing

This type of testing is a formal test conducted to determine if the SW satisfies its acceptance criteria and to enable the customer to determine if can be accepted. (Melnik, Meszaros 2009) The acceptance criteria describe customer's requirements and are written according to the Requirements Specification Document. Acceptance testing is a "black box" execution and can be done automatically. It is allied to system testing.

As conclusion, acceptance testing:

- demonstrates if all customer's requirements are satisfied
- users are inherent part of the process
- usually units with the system testing

- is done both by customers and testing team
- is done using real system and emulation

2.6.5 Regression testing

Regression testing is the process of validating the modified SW to detect whether new errors have been introduced into previous tested code and to provide confidence that modifications are correct (Graves et al. 1998). The modification can cause defects not only in those units, which have been changed, but in whole program. As usual the developers create an initial test suite, and then use it for testing the SW after modification. The simplest regression testing strategy is retesting whole program by rerunning every test case in the initial test suite. However, this approach is expensive rerunning all test cases requires an unacceptable amount of time. An alternative approach is regression test selection, which means rerunning only a subset of the initial test suite. Of course, this approach is cheaper, but it has disadvantages. Regression test selection techniques can have substantial costs too and discard test cases that could reveal faults. So, regression test selection reduces fault detection effectiveness. The main problem of the testing is choosing trade-off between the time required to select and run test cases and the fault detection ability of the test cases. The chosen regression test selection algorithm significantly affects the regression testing cost-effectiveness (Graves et al. 1998).

Several regression test selection techniques have been investigated in the literature. Here are some examples. First of all, Retest-All Technique, which reruns all test cases (Rothermel, Harrold 1996). It may be used when test effectiveness is the utmost priority with little regard for the cost. According to Random/Ad-Hoc Technique testers often select test cases randomly or depend on their experience or knowledge (Graves et al. 1998). Minimization Technique suggests to select a minimal set of previous test cases that covers all modified elements (Hartmann 1990). And of course, some combinations of those techniques can be used.

As conclusion, regression testing:

- performs when program has been changed
- can rerun all test set of previous test cases (expensive)
- selects a part of previous test cases according to different techniques

There are other different types of testing such as stress, scalability, and many others, which are implemented during different development steps.

2.7 SW development models

The SW process models describe phases of the project development. The first models appeared in the 1950's and 1960's. (Cornett 2011) The SW life cycle models are used to create a conceptual basic algorithm for optimal management of the SW systems development, which should be a basis for organizing, staffing, coordinating, budgeting, and directing the SW development activities. (Scacchi 2001) Since the 1960's, many models of the SW life cycle have appeared. The most common of them (the waterfall, V- and spiral models) are described in this section.

2.7.1 The waterfall model

One of the first SW process models that have been widely used is a waterfall model. This model was developed in the time of simple programs. That time is described by simple program requirements, so, the program could be tested inserting it into the card reader and observing its releasing. It allows dividing the first time SW engineering into phases. The basic concept of the waterfall model is shown in Figure 2.1.

It consists of different steps. First of all, Requirements Specification Document, which includes all requirements that should be fulfilled by the program, is generated. These requirements are determined according to user's needs. Then System Design should be done. The system has to be properly designed before implementation. This step includes different designs. For example, architectural design that describes the main components of the system such as definition of a computer platform and an operating system is done to define HW. The main purpose of this phase is generation the System Architecture Document. During SW design step the SW blocks that were described in previous step is

transformed into code models. The interfaces and interactions of the modules and the model functional contents are defined. The output of this phase is a SW Design Document.

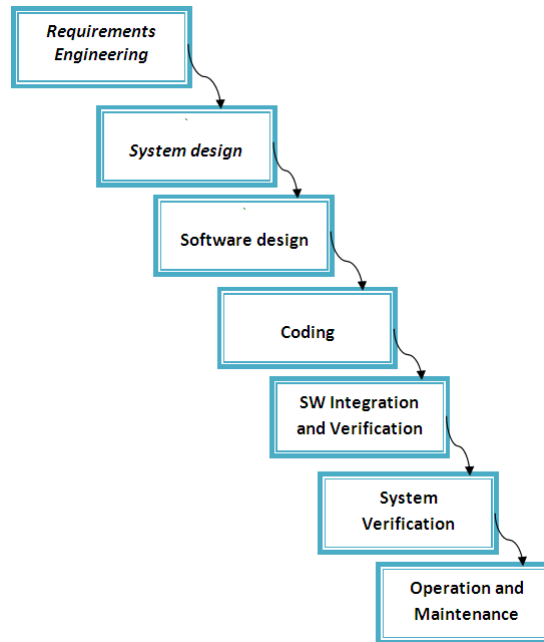


Figure 2.1 The waterfall model of SW development models. Adopted from (Target)

Coding step means the actual coding is started based on the SW Design Document. SW Integration & Verification step includes unit and integration testing. System Verification means system testing, which proposes explosion of whole system, including testing of the original HW and environment. If the SW passes all tests, the customers will get it and the operation will begin. All the problems which were not found during the previous phases will be solved in this last phase. It can be never ending phase.

This model has many disadvantages, which are not critical for small programs, but cause many problems for complex systems. One of the main critical parts is the first step that proposes defining of all requirements. Usually, only a small part of the requirements is known at the beginning and it is necessary to change them during the project. On the other side, the process only allows for a single run through the waterfall. This model permits iterations only inside one phase, which delays problems solving. As result, all problems are solved during last steps. It causes a bad program design and low quality. Also the huge last phase "Maintenance" becomes the most important part of design and takes much time.

As programs became more complex the waterfall model was not enough. Programmers found it more and more difficult to work with this model and other models such as V Model were created.

2.7.2 The V model

V model is a future development of the waterfall model. The basic concept of the model is shown in Figure 2.2. The steps of the process are almost the same. Instead of going down the waterfall in a linear way the process steps of the V-model are bent upwards at the coding phase, to form the typical V shape. One of the reasons of such form is a duplicate in the testing phases for different phases.

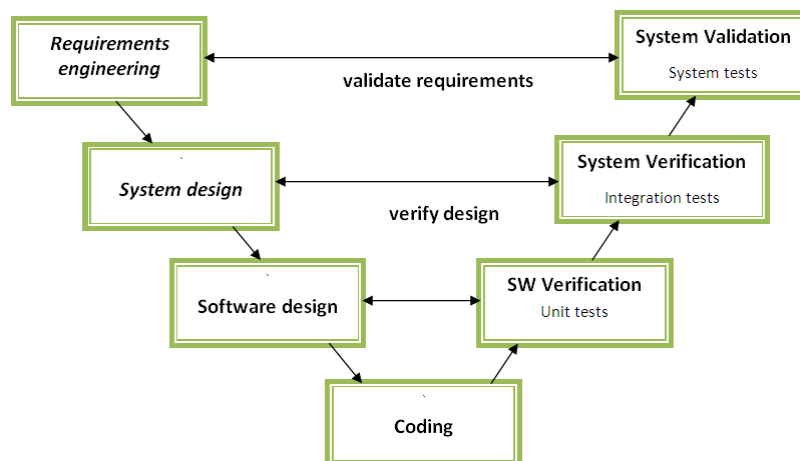


Figure 2.2 The V model of SW development process. Adopted from (Melnik, Meszaros 2009)

The V-model proposed clear definition of different testing levels. Test cases are created by customers in the form of requirements. This model allows testing different units individually.

Another advantage of the V model is replacing "Operation & Maintenance" phase with the validation of requirements. It means that during the last phase not only the correct implementation of requirements has to be checked but also correctness of the requirements should be checked. Instead of ending maintenance phase the V-cycles was defined. It means that if during the validation steps it became obvious that requirements are not full or

are incorrect, the modification of the problem began from the first stage. So, two or even more V-forms can be realized during the design process.

The V-model is not ideal too because it can be developed in sequence of great number of v. Thus, some different models of the SW process design were offered, for example, the spiral (Target) and agile models.

2.7.3 The agile model

Agile methodologies represent a successful, modern method by which the SW can be developed. (Maher, Kourik & Chookittikul 2010) Those methodologies become more and more prominent in the SW industry because of its flexibility.

The agile methodologies have some special characteristics that focused on simplicity and speed. First of all, SW team is testing developed SW uninterruptedly. New releases of the SW are produced at frequent intervals usually two times a month. The other main idea of the method is keeping code as simple as possible and at the same time technically advanced. It lessens the needed documentation. Also the agile model supplies close communality between the SW developers. It is addressed to the boosting team spirit. On the other hand, relationships between clients and developers are settled over strict contract. Agile cooperation between those groups reducing the risks of non-fulfillment regarding the contract. The special development groups than comprise both the customer and the SW developers are organized. The quality of the product increased if such groups are well-informed, competent and authorized. It means that participants are prepared to make changes and improvements in the product. Those peculiarities are addressed to make the development process easier, high quality, and flexible.

Although there are many variations of agile models such as extreme programming (XP), scrum, crystal family of methodologies, feature driven development, and many others, all of them propose the cooperative, incremental, cooperative, straightforward, and adaptive SW development process. (Abrahamsson et al. 2002) Extreme programming (Beck 1999) and scrum are two of the oldest and widespread agile methodologies

Nowadays, the SW models must account the interrelationships between the SW products and the production processes, tools, people and their workplaces. Consideration of the factors can utilize features of the traditional SW process models, such were described previous. Thus, new agile models are under intensive research.

2.8 Conclusion

Testing is the main criteria of the SW quality, but not a tool to prove if the SW works or not. Historically, testing was implemented only during the last design steps, when there is no opportunity to embed cardinal changes. Therefore, the product quality decreased. Testing discrimination between levels permits to have iterations during the design process. Hence nowadays testing is implemented during all development process as early as possible. Thus, huge changing of the project, like adding new models and requirements correction, became possible without program design deterioration.

New SW development models focus on the developer and customer intercommunion to reduce the risks of non-fulfillment regarding the contract, flexibility and uninterrupted program revising, testing and improving. Such agile methods are becoming more and more popular, especially in small develop teams. The main idea of these methods is keeping code as simple as possible and at the same time technically advanced.

Testing became one of the most important design process parts that requires particular design. So, nowadays represents a set of operations that must be done. It consumes at least half of the labor expended to produce a working program (Beizer 1990). So, it stays the hugest part of the SW design. Therefore, it is important to optimize the design step that involves reducing all kinds of charges. The optimization can be reached by clear and short documentation, using testing for different cases, correct recording results of testing and, the most potential tool, automation tests. Of course, only complex reasonable implementation of those actions can give good results.

3 TESTING AUTOMATION

Automated testing is one of the most effective tools for reducing material and time costs of the testing process. In the middle of 1980 the first tools for automated testing appeared and the automated process was implemented in different kinds of SW testing. In a short time, it became obviously, that automated tests can cause many problems such as increasing product cost, decreasing reliability and even project fail.

Nevertheless, automated testing popularity has been increasing constantly, because of the huge amount of advantages that are involved by the automated process implementation. First of all, it reduces the human contribution to the work that means decreasing of human errors. Also automated testing abates the testing coast and, in that way, the final product cost. For example, the costs for tester's training and motivation become lower.

Moreover, the automated testing leads to the essential time saving. For example, one simple case is examined. A test script, which contains of the ten inputs and twenty outputs that include one required result, is executed. The average time requirements for manual and automated testing are shown below. So, the manual testing require about 100 second for one test execution (data input – 50 seconds, results – 2 seconds, looking for necessary information -15 seconds, checking results – 30 seconds). On the other hand, automated testing of such easy case requires about 4 seconds, because it takes about 1 second for data input, 2 seconds for getting results, 1 second for finding needed information and checking it. Thus, the automated testing is about 25 times faster than the manual testing in quit simple cases and even faster in different cases.

Also automated testing prevents many human errors, for example, mixed data inputs. It allows running long time testing such as usability testing during all day long even during night time that causes reducing of time costs. So, the implementation risk of test automation is justified.

This chapter familiarizes with the basic principles of automation testing, describes different approaches of automation test data generation, oracle and define main problems of automation testing.

3.1 Manual and automated tests

The main ideas of manual and automated testing should be compared to understand the differences between these processes.

In Figure 3.1 the scheme of manual testing is shown. Pink boxes present necessary documentation; orange boxes contain actions that are done manually. The informal requirements are used to create formal specification during the requirements analysis. SW code writing is based on this specification. When the code or one of its logic parts is ready testing should be done. To create the test cases some analysis is required. It can be static analysis (Figure 3.1) for “white box” testing or requirements analysis for “black box” testing. Manual testing proposes that these operations are made by people. The ready test cases are sent to the SUT and oracle. This SW produces some outputs. Oracle is used to create an etalon outputs usually according to the special tables or the program specifications that are made by the develop team. At the final stage tester (person) compares the actual and etalon outputs and makes decision if the SW passes the tests or not.

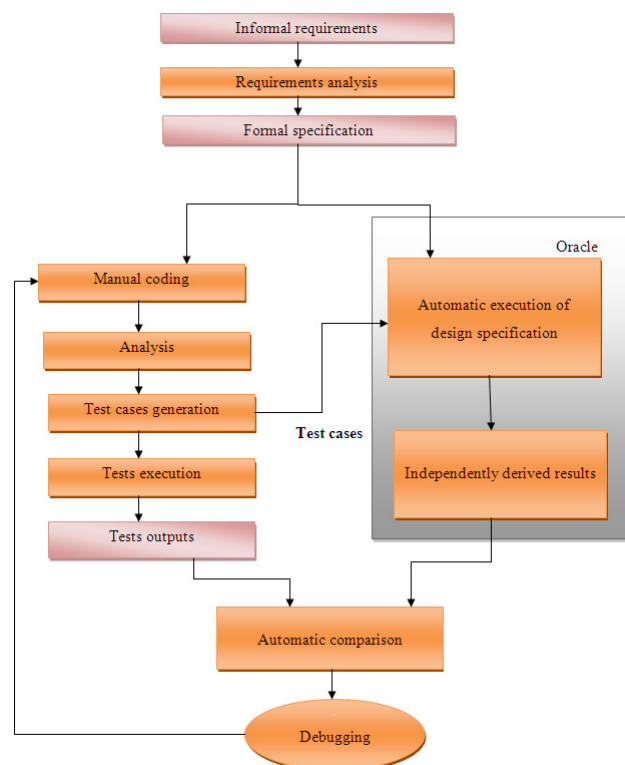


Figure 3.1 Principle of manual testing process. Adopted from (Brown, Roggio & McCreary 1992)

Most of the testing activities can be automated. The principle of testing automation is shown in Figure 3.2. The only manual operations after formal specification producing, which is doing during close developers and customer communication in general case, are coding and debugging. All other operations are automated (green boxes in Figure 3.2). The formal specification goes to manual coding and oracle that proposes its automation execution. The test cases are generated automatically based on the specifications and the criteria. After test cases execution the SUT and oracle outputs are compared and a fail report is created. SW debugging starts according to this fail report.

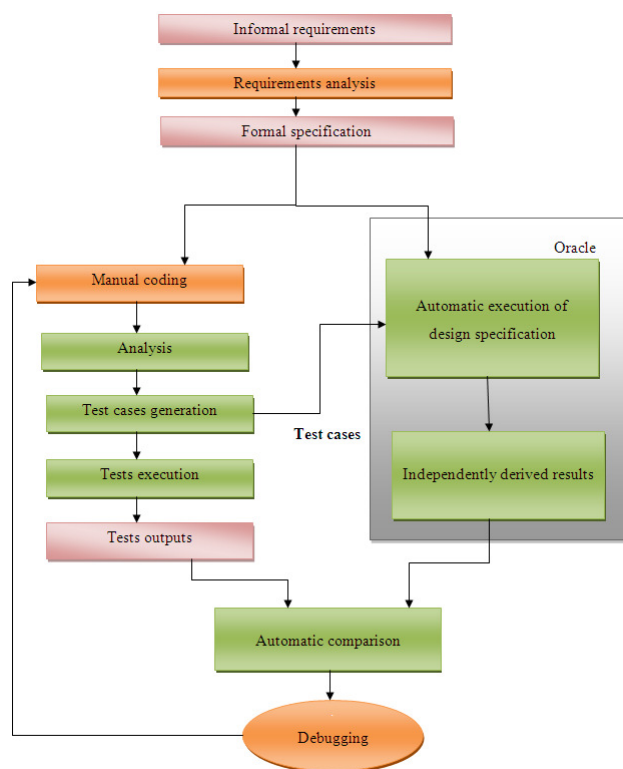


Figure 3.2 Automation testing. Adopted from (Brown, Roggio & McCreary 1992)

Obviously, different cases have limits and it is impossible to implement complete automation in most of cases. As usual, only some parts of the testing process are automated.

3.2 Test automation concept

A test automation system requires different components depending on its type, such as test oracle, input data, a test driver and a test harness. These basic components are illustrated in Figure 3.3.

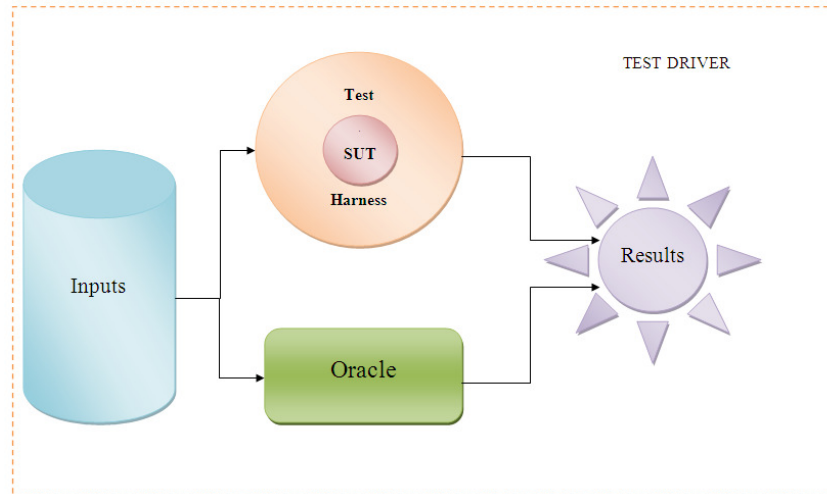


Figure 3.3 Parts of automated testing. Adopted from (Kanstrén 2010)

The test execution process is controlled by test driver controls. Test harness is a set of tools controlling the creation, maintenance and execution of the test cases. (Bartels et al. 1994) A test harness isolates the SUT from different environment parts for different testing purposes. It has several roles including setting up the initial state of the SUT for each test and setting up the testing environment. (Kanstrén 2010)

Another important part of the testing system is test input that can take different forms, for example, message sequences. Test inputs are created by programmers or generators during the process that is called test data generation. In SW testing test data generation is the process of identifying input data that satisfies selected testing criterion. (Korel 1990)

Inputs have many combinations and their effects on the SW behavior must be tested. Obviously, in complex system it is impossible to execute whole set of the test input combinations. Data input comprises only some part of the possible test inputs depending on the test purposes and can be created manually or automatically by generators. Manually crafting a good input data set for this is difficult and time-consuming. On the other hand, the generators can automatically create large quantities of different data types using different approaches. Some of the most widespread automated test data generation

approaches are traditional such as *random* (Duran, Ntafos 1984), (Ciupa et al. 2008), *symbolic* (DeMillo, Offutt 1991), (McMinn 2004) and *actual* execution (Korel 1990) and search-based optimization techniques that enable to generate a set of data for a specified goal (McMinn 2004). Search-based techniques use different optimization algorithms such as hill climbing, genetic algorithms or simulated annealing (Metropolis et al. 1953). During last decades other automation SW test data generation approaches such as domain-specific (Bertolino et al. 2007), (Yaun, Memon 2007) and program-invariant-based (Pacheco, Ernst & Eclat 2005) have been becoming more and more popular.

A test oracle has the same functions as in manual testing, so it is used to verify correctness of the received output according to test inputs. The problem of the oracle automation does not get as much attention as, for example, automated test cases generation. Such simple suggestions as made test oracle construction by manually translating post conditions (Bicego et al. 1986) or formal explicit specifications are not spreaded because can handle (Adrian, Branstad & Cherniavsky 1982) only nondeterministic results (Brown, Roggio & McCreary 1992). New complex methods that are based on, for example, artificial intelligence (Kanstrén 2010) or N-version diverse systems (Manolache, Kourie 2001), are more useful, because they propose automated oracle process.

So, the test automation is a difficult and complex process that proposes first of all automated input data generation, automated oracle and automated test execution (Figure 3.3). To make test inputs data automatically is possible in many cases; on the other hand oracle automation requires much work and a long period of time.

3.3 Automation testing metrics

The complexity of the automation makes its management process very difficult. Therefore, before stating the process the detail plan should be developed. Otherwise, team's lack of inexperience in developing and testing can transform automation testing in a never-ending process. Even in well planned projects defects that bring the project at the first step can be defined. So, it is very important to control the automation process. Different kinds of metrics are good tools to define clearly the phase of the testing automation. Good metrics are objective, measureable, meaningful, simple, and have easily obtainable data. (Garrett

2011) Such metrics as percent automatable, automation progress, and percent of automated testing coverage satisfy the requirements and can be used to supervise automation testing development.

3.3.1 *Percent automatable*

The process of automation testing is complex and laborious and does not always return the expenses. So, it is very important to evaluate future benefits of automation testing. The benefits are defined by the testing SW and the testing process itself. Last one depends on many conditions and has different automatable degree. It means that not all tests are able to be automated. Therefore, in the beginning of any automation testing project that intent to automated manual tests or improve test automation percent automatable (PA) should be determined (Eq. 3.1).

$$PA = \frac{ATC}{TTC} \cdot 100\% \quad (3.1)$$

where PA - percent automatable, %; ATC –number of test cases automatable; TTC - number of total test cases. (Garrett 2011) PA describes how many test cases of all specified test cases are automatable or, in other words, can be done by automatic way.

To calculate this metric, first of all, the test cases should be divided into automatable and not automatable cases. This property of the test is difficult to define because if there is no material and time limits, almost all tests are automatable. Of course, there are standard cases that are easy or impossible to automate. No automatable cases are, first of all, cases that are under design, in flux or not stable. So, the dividing process requires careful individual approach.

PA is employed to set the automation goal, but it does not always adequately describe the testing process. Often if the test can be automated, it does not mean that this test should be automated. It can demand much time and money, but achieves no goods. Therefore, this metric is used in package with other metrics.

3.3.2 Automation progress

Automation progress (AP) is another significant metric that shows the percent of the automated test cases at the specified time compare to all automatable test cases. The metric can be calculated according to Eq. 3.2 (Garrett 2011)

$$AP = \frac{AA}{ATC} \cdot 100\% \quad (3.2)$$

where AP is automation progress, %; AA is a number of actual automated test cases; ATC is a number of automatable test cases.

This metric helps to define the phase of the automation process. The goal is to automate as many test cases as possible. The number of the test cases is accepted to be 100%. So, in the beginning of the project this metric is 0% and at the end of the automation process it is 100%. Obviously, it is tracked over time. In the begging it changes slow, because the developers are not familiar with the testing tools, the SW and they have to do many other difficult things except the cases automation. Then the process usually has linear dependence on the time. It is one of the most productive phases of the automation process, when many technicalities are settled and development team is focused on the automation. During the last stage, when all easy and obvious test cases had been automated, new problems such as different cases break the development process.

3.3.3 Test progress

Test progress (TP) is often associated with the automation progress, but actually it is two different metrics. The AP can be used only during automation testing, whereas the TP is useful even during manual testing. This metric shows testing progress over time. It is defines as the numbers of test cases that are completed during the specified period of time (Eq. 3.3). (Garrett 2011)

$$TP = \frac{TC}{T} \quad (3.3)$$

where TP is test progress, number of test cases per time; TC is a number of completed test cases; T is some unit of time (day, week or month).

3.3.4 Percent of automated testing test coverage

This metric indicates the test coverage that is achieved by the automated testing. In other words, it describes testing completeness. The coverage of the product functionalities can be determined using Eq. 3.4

$$PTC = \frac{AC}{C} \cdot 100\% \quad (3.4)$$

where PTC is percent of automation testing coverage, %; AC is automation coverage; C is total coverage. (Garrett 2011)

The metric is good to describe the automated testing process, because it does not measure the number of automated testing, but its quality. For example, if the one hundred tests that execute the same paths are automated the percent of automated testing test coverage is low. On the other hand, if one test covers fifty percent of all testing area, this test automation increased the metric significantly. Therefore, the percent of automated testing test coverage indicates the testing dimension.

3.3.5 Defect trend analysis

Defect trend analysis closely relates to the defect density. It shows if project is improving or the situation is going worse, so it describes the project health. It is calculated as (Eq. 3.5)

$$DTA = \frac{D}{TPE} \quad (3.5)$$

where DTA is defect trend analysis; D is number of known defects; TPE is number of test procedures executed over time. (Garrett 2011)

Effective defect tracking analysis presents a clear view of the testing status throughout the project. (Garrett 2011)

3.3.6 Defect removal efficiency

The testing process is one of the most important tools to evaluate quality. So, it is possible to evaluate the testing process using the product quality. Defect removal efficiency (DRE) is one of the most popular metrics for quality tracking. It is not specific to automation, but its combination with automation efforts gives good results.

This metric is used to determine the effectiveness of the efforts that are used to removal the SW defects. Also it is one of the indirect product quality measurement. The value of the DRE is calculated as (Eq. 3.6)

$$DRE = \frac{DT}{DT + DA} \cdot 100\% \quad (2.6)$$

where DRE is defect removal efficiency, %; DT is a number of defects found during testing; DA is a number of defects that are founded after delivery. (Garrett 2011)

The high DRE value means that the defects were founded and eliminated in time, during first design steps. So, the product has high quality. The best DRE value is 100% that cannot be reached in practice.

This metric should be calculated during all design steps to expose how the defects are lighted in the different design phases. Also it can be calculate after released products as a measure of the number of the product defects that were not caught during the product development and testing.

There are other different metrics than can be used for automation testing process evaluation such as defect aging, defect fix retest and many others, but they are not uses so often like seven previous. (Garrett 2009)

Metrics are an important indicator of the SW quality and automation testing progress. There are many metrics that describes the automation testing process from different points of view such as quality, coverage and progress. Automation testing development is the compound process that can be fully described by the system of different metrics.

3.4 Main test automation problem

The most people imagine the test process as a sequence of actions. In fact, the testing process can be described as a sequence of interactions interspersed with evaluations (Bach 1996). The interactions can be predictable, but most of them are specified, complex and equivocal. Nevertheless, such approaches to conceptualize a general sequence of testing actions can be useful if the main purpose is reducing testing to rote sets of actions. But even in the situation the result can be shallow and limited. On the other hand, manual testing has such property as adaptability that means easy changing according to the new circumstances. So, humans do not require a strong sequence of actions to reveal many defects and to distinguish them from harmless anomalies, which is a great advantage compare with automation testing. Therefore, automation testing is the best solution for a narrow spectrum of testing.

Other common misconception says that testing means repeating the same actions over and over. Honestly, if no one bug was founded at the first test case execution, the bugs will be revealed in other executions only if new bugs are implemented in the SW. At the same time manual testing always has variation of the test cases and it provides higher percent of new and old defects detecting. Variation is one of the great hand testing advantages (Bach 1996). Thus, the number of program testing execution will bring results only if test cases are variated.

Against to the widespread opinions, not all testing actions can be automated. Some tasks are very easy for humans, but at the same time they are difficult for computers. For example, interpreting of test results is the hardest part for the automation process. The current topical SW is innovative, which means they are high degree of uncertainty that compounds the automatability problem of the SW testing. Also the project are developed using incremental approach that involves different kind of fundamental SW changing even at the last design steps, which compounds the automatability problem too. Thus, automation testing can easily transforms into a slow, expensive, and ineffective process that contradicts other widespread opinions “an automated test is faster, because it needs no human intervention”. (Bach 1996) This expression is wrong not only because automation testing can be slower, but it always requires human intervention. Such process as analyzing testing results and fixing bugs are always done by people. It is impossible to imagine

testing without hitches, therefore it is impossible imagine automation testing without human intervention.

The automation reduces some human errors. It is the perfect decision to implement a long list of secular tactile actions, which is difficult for people.

Most of managers want to quantify the costs and the benefits of automation testing compare to manual testing after implementation of new strategy. Unfortunately, these two processes are really different. So, the kinds of testing have different concepts, dynamics, they were created to reveal different kind of the bugs. Therefore, direct comparison of them in terms of the dollar cost or number of bugs found is meaningless (Bach 1996).

An expectation of "significant labor cost savings" that is caused by automation testing is a myth too. The cost of automated testing is comprised of several parts such as the cost of developing the automation, operating the automated tests, maintaining the automation as the product changes, and costs of any other new tasks necessitated by the automation. (Bach 1996) Comparing total cost of the actions to the manual testing shows that automation does not reduce the cost of testing because it does not decrease testers' work to such an extent that the manual testers do not have work to do.

The cost of testers work depends on many factors including the cost of the used test tools, the test developers' deftness, quality of the test suite and many other different factors. Obviously, writing a single test involves less effort than creating a test process for huge system that includes choosing automation testing tools, coordination automation to the rest process, and many other actions. This process can take months of team hard work or even more. Also the automation costs include the costs of special new tasks, which are required by automation. Hence, automation testing is an expensive action that includes charges for test cases documentation, testing of the automation process, checking and analyzing test results, changing test codes in the case of product changes, organize the development process of the automation and many others. All those action need people active participation. So, automation testing is not a good tool for reducing labor cost.

The probability of harming test project by automation testing is not taken seriously. Unfortunately, wrong automation design and implementation can cause huge numbers of problems. The full understanding of SW performing is required for well-done automation.

Automation could be transformed in a large mass of the test code that is not fully understood by testing teams if there is not clear test strategy.

As conclusion, it should be said that

- testing is not a sequence of actions, but a sequence of interactions interspersed with evaluations
- testing does not mean repeating the same actions over and over, but requires some variations
- not any test can be automated
- automated testing reduces the tester action errors, but does not reduce design test errors
- automated test is created not to substitute manual testing, but to supplement it
- automated testing decreases the cost of the testing process, but increases the cost of test development
- the full documentation is required during the test automation process

3.5 Conclusion

Manual and automated testing are only skills, but very difficult skills. The main differences between these processes are shown in Table 3.1.

Table 3.1 Manual and automated testing

	Manual	Automated
Data input	Flexible data input allows using different values during different test cycles to make test cover higher	The input data is strictly set
Result analysis	Flexible; tester can check different fuzzy criteria	Strict, fuzzy criteria can e tested using etalon model only.
Repeatability	Low; human factor and flexible data input cause the non repeatable test cases	High
Reliability	Low, the long test cycle lead to reducing of tester's attention	High, does not depend on the length of test cycle;
Sensitiveness to insignificant product	Depends on the process description, usually tester is able to test product if its form and functions have been changed a	High, the little product modification causes great automated test changes;

modification	little bit	
Velocity of test execution	Low	High
Ability of test generation	Low, the low velocity does not allow to execution test cases	Can be

The test automation has great promise and brings significant benefits, such as better resources using, testing of attributes that are difficult to test manually and increased confidence. At the same time automated testing has its limitations. First of all, it does not replace manual testing. Moreover, the manual testing finds more defects than automated one. Also the correctness of expected outcomes has higher reliance. Test automation does not improve test effectiveness. Automated quality is independent on test quality The testing tools have no imagination and are not very flexible. On the other hand, test automation can significantly increase the SW testing productivity and quality. (Fewster, Graham 2008)

4 CRANES

Crane SW development such as any other industrial SW growth is standing aside from the other kinds of the SW development processes. The bridge crane SW development depends on many factors. The main purpose of the SW is to propose safety crane operation. So, such SW cannot be elaborated without the basic knowledge about the controlled object, its main parts, functions and behavior under different conditions. Modern bridge crane has many functions and safety issues that transform the SW in a huge complex system.

Moreover, the crane control system consists of the SW and the HW that influence on each other and cannot be developed or examined separately. It complicates the SW development and testing process and causes adopting of different approaches such as SW and HW simulations. Also it requires especial skills from its developers. So, the industrial SW developers should be experts not only in programming, but, first of all, in the process they are going to automate. Also the knowledge in such areas as analog and digital circuit technology, microcontrollers, automated control theory, industrial electronics, and many others is required.

This chapter describes the modern bridge crane main parts, functions, the control system and SW architectures.

4.1 Crane. Component parts. Functions

Crane is a device for lifting and moving heavy weights in suspension. (Anonymous b) An electrical overhead travelling (EOT) or a bridge crane is one of the most popular industrial cranes that are used for lifting various items and materials. This type of cranes comprises of the several components such as a girder, a trolley, a hoist, end trucks, a cabin or a control device, electrical control, a trolley cable system and a service platform. Some of these main parts can be seen in Figure 4.1.

The bridge cranes propose one or several girders according to the load. The most common type of these cranes is a single ginger crane. However, cranes with double, three and even four gingers are used to hand heavy loads. Also in this case several auxiliary trolleys that

provide easy handling of the complex form load can be necessary. The girder is stand on the rails. The trolley that holds the hoist and its motor is placed on a girder or between two girders. The electric cabinets are placed on a girder as well.

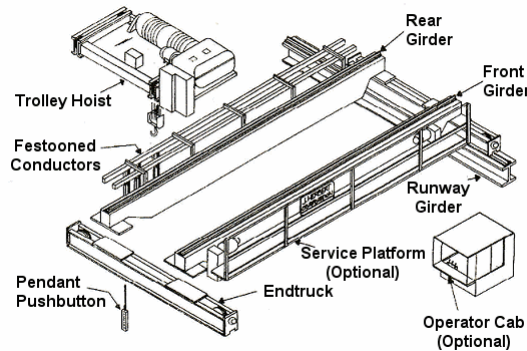


Figure 4.1 The main crane component parts. Adopted from (Anonymous)

Crane operation proposes such functions as hoisting up and down, moving trolley right and left, moving crane forward and backward. EOT crane can be operated by operator from a cabin, a factory floor or a control room by radio, infra red or pendant control. The infra control is used in the outdoor cranes. The friendly frequency license politic makes the radio control possible.

Personal safety is one of the main characteristic that is taken into consideration during the crane design and implementation. Such precautionary measures as load weights displays, warning horns and work area lights are used. Some manufactures as harbors cannot provide a safe place for the operator, so the special cabins with cooling and heating devices that is placed usually under a girder are required. The crane safety is increased by the implementation of the wind meters, motors, electric cubicles and gearboxes heating, rail claps for outdoor cranes.

The safety and smart crane operation are required a great complex control system.

4.2 Crane control system

The EOT cranes are widespread, so there are many kinds of the cranes that are distinguished by their mechanical and electrical parts. The present work focuses on the

modern EOT cranes, which have control system based on PLC (Programmable Logic Controller).

The crane control system consists of three main parts (Figure 4.2). The electromechanically part is represented in blue color in Figure 4.2 and is responsible for all safety functions to maximize their operational reliability. The crane PLC (orange color) is a part that focuses on the main crane function proposition. This part of the crane architecture is described deeply below. Special functions as control of conveyor warehouse gate are implemented using the hoist control place (violet color). The interaction between PLC and host system can be obtained on different field buses.

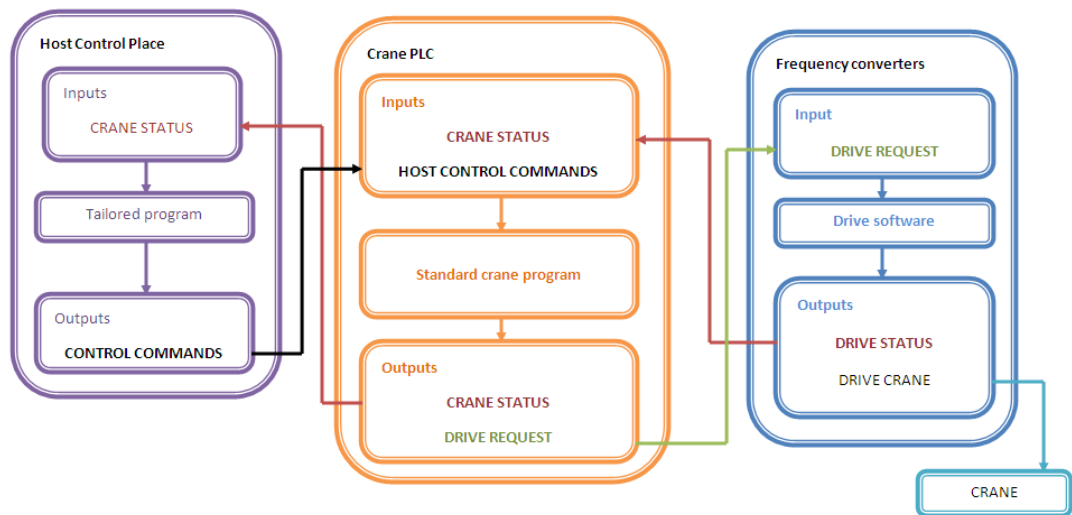


Figure 4.2 The crane control system

4.3 The PLC based crane control system

The modern crane PLC control system architecture is presented in Figure 4.3. The operator manages the crane using the control station from the ground level. The PLC receives the signal from the radio receiver and forms the output signal that goes to the hoist, the trolley or the bridge frequency converter on the field bus. Also this bus is used to transmit information from the electro mechanical crane parts to the PLC for collecting monitoring data to maximize the operational reliability of those parts.

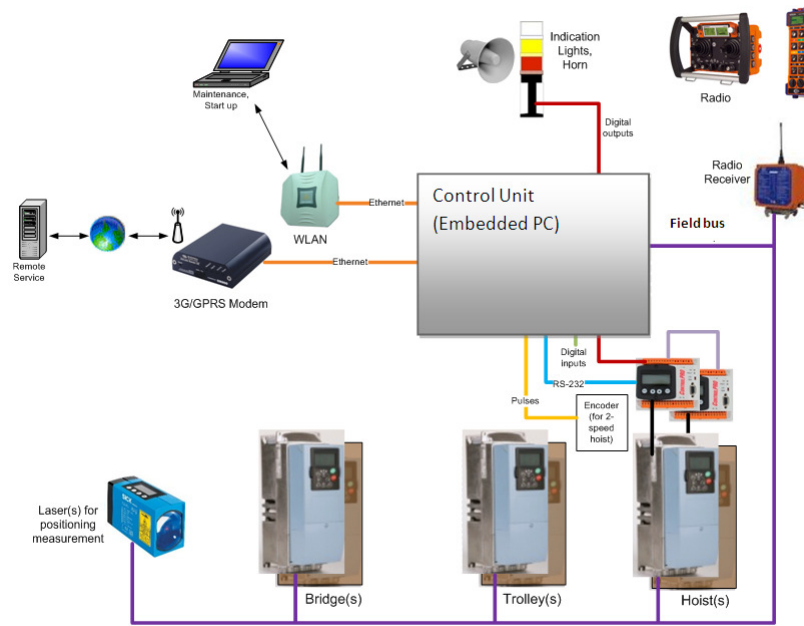


Figure 4.3 The crane control system

Position measuring devices are necessary to determine the position of the different crane parts, which is important for the control system during positioning or moving synchronization of two or more hoists, trolleys or bridges.

Remote service is used, for instance, for the crane monitoring and information collecting, and is connected with the PLC through the global network.

Hoist drive supervision unit is implemented to monitor and to limit hoist movements. This unit recognizes drive requests and, if the drive allows, this request will be given to the drive. Usually, only the hoist overload can block driver request, so the interlock report is sent to the PLC, where it is stored and displayed to the operator.

The PLC program should provide specified operation functions during different crane modes (service factory testing, factory test bypass, service, and normal modes), collects monitoring data from safety functions, and gives a message from these events through the service interface.

First of all, the PLC is responsible for the main functions of the different crane part, such as a hoist, a trolley and a bridge. The hoist is one of the most important crane parts. It provides load up and down movement. The chain hoist that is used in the modern EOT cranes can be seen in Figure 4.4. The hoist control system embraces direction and speed

control, overloads protection, common several hoists using and hoist movement interlocks to avoid bridge, floor or personal hitting by hoist and load, emergency situations.



Figure 4.4 Wire rope hoist

The trolley includes a hoist and its machinery. Also electrical equipment of the trolley itself is located there. End carriages, the main purpose of which is to transfer the crane end to distribute the load stress from the main girders to the crane rail, are a part of trolleys equipment as well. A rope drum is designed to last the lifetime of the hoist and is a sign of very heavy duty operation or misuse. The main trolley function is moving hoist and load to the right and left directions.

Bridge is a crane part that carries a load, a hoist, a trolley and their equipment. The main bridge function is moving trolley with the hoist, hook, and load to the forward and back directions.

The trolley and bridge control systems have much in common. These crane components require, first of all, speed (open loop control) and direction control systems, common use, collision prevention and different interlocks.

Sway control system is one of the most important crane systems that provides such crane movement that sway of the load is minimized, so the load movement path on the main hook stays linear and stable. The sway system is responsible for the safety, so defects in this system forbid any travelling actions of the trolleys.

Crane is equipped with indication horns and beacons to inform operator and personal around the crane working area about crane operation. The operation of those devices is controlled by the PLC as well.

Also the PLC control system provides general functions such as crane on/off control and user interface functions, for example, machinery selection function, and overload protection.

The main crane functions that are controlled by the PLC are described deeper in Table 4.1.

Table 4.1 Main crane functions

Equipment	Functions	Interlocks
Hoist	<i>Direction control</i>	
	• load hoisting	<ul style="list-style-type: none"> • hoist overload (110%); • bridge overload; • up stop limit; • rope over wrap limit; • working limit
	• load lowering	<ul style="list-style-type: none"> • down stop limit; • working limit; • slack rope active; • overload of one hoist during tandem using;
		<ul style="list-style-type: none"> • drive fault; • field bus fault; • working limit mode; • active of both drive request; • hoist motor over temperature;
	<i>Speed control</i>	
	two step speed; five step speed with ramps; 255 speed steps from joystick	<ul style="list-style-type: none"> • hook position error; • over wrap fault during up; • in common use slowest hoist speed reference;
	<i>Common use</i>	
	<i>Direction control</i>	
Trolley and Bridge	forward and reverse moving	<ul style="list-style-type: none"> • stop limit; • drive fault; • field bus fault; • working limit mode; • sway control request; • bridge overload;
	<i>Speed control</i>	

	two step speed with ramps, five step speed with ramps; 255 speed steps from joystick	<ul style="list-style-type: none"> • hoist/trolley position error; • limit position reach; • anti-collision system request; • during service mode;
	<i>Common use</i>	
	<i>Anti-collision system</i>	
	<i>Sway control system</i>	
	<i>Skew control system</i>	
User interface	<i>Control place</i>	
	<i>Host control place</i>	
	<i>Control place selection</i>	
	<i>Machinery selection function</i>	
Indications	<i>Indication horn</i>	<ul style="list-style-type: none"> • overload; • movement starting; • finish position;
	<i>Warning beacons</i>	<ul style="list-style-type: none"> • lit: crane is on and sway control is active; • blinking: service mode is selected
	<i>Automation beacon</i>	<ul style="list-style-type: none"> • lit: end position can be started; • blinking (2 sec) hook is on the target; the semi auto position is active;
	<i>Alarming beacon</i>	<ul style="list-style-type: none"> • lit: active fault in the PLC system ; • blinking (1 Hz): thermal alarm in the hoisting motor; alarm control system action; • blinking (2HZ): crane overload.

The PLC is responsible for the correct operation of the different devices such as the hoist and trolley drivers, indication systems and many others. At the same time some points of the crane and personal safety is defined by the crane PLC program too. So, this system requires fast response to any event. Therefore, the PLC SW must be complete and reliable and, at the same time easy to understand, service and modify. All this strict requires transform the crane SW development and testing into very important, intricate, and responsible processes.

4.4 Crane software architecture

This thesis focuses on the EOT cranes that are based on PLC. The SW development environment system turns almost any compatible PC into a real-time controller with a multi-PLC system, NC (numerical control) axis control, programming environment and operating station. The system allows data exchange with different applications, for example with Office programs. Moreover, the SW ensures the highest deterministic features of the program executions, independently of other processor tasks. It allows creating a complex and complicated SW system, which can be seen in Figure 4.5.

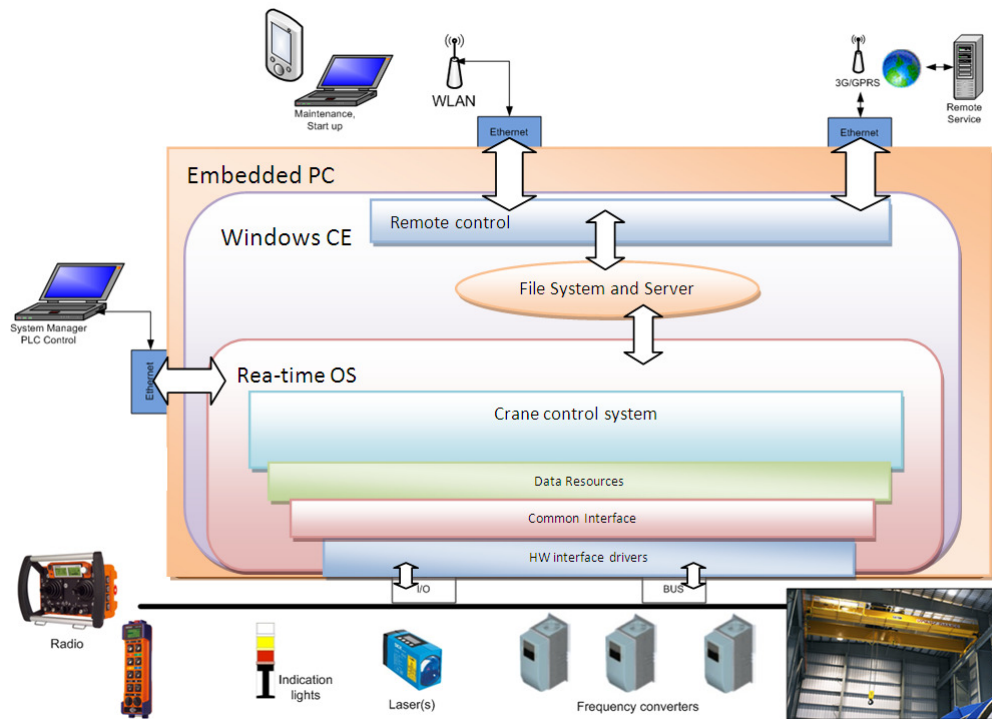


Figure 4.5 Crane SW architecture

This crane SW consists of several levels. The first level is represented by Windows CE (Consumer Electronics). This level comprises of the environments for programming, diagnostics and configuration and provides system communication remote and maintenance services. The second level includes the run-time system that is responsible for the real-time control programs execution. The real-time system contains the crane control SW, HW interfaces drivers and interfaces for different data and message creation, storage and communication.

The PLC is programmed in accordance with IEC 61131-3 independently of the manufacturer. It supports all the standard IEC 61131-3 languages and has a convenient editors and a fast compiler. Incremental compilation prevents long turnaround times. The program code can be changed during the PLC running.

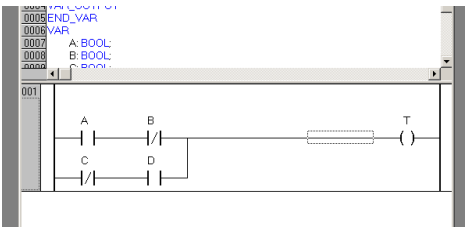
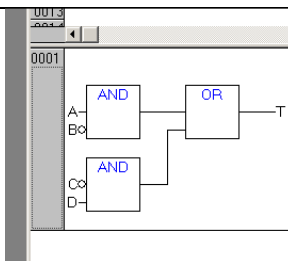
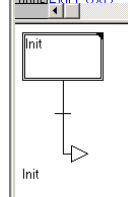
The design of IEC 61131 standard was started in the 1979 and in the 1982 the first part of the product that is combination and continuous of different standards. (John, Tiegelkamp 2010) Nowadays, this standard consists of five programming languages, which are Ladder Diagram (LD), Instruction List (IL), Function Block Diagram (FBD), Structured Text (ST) and Sequential Function Chart (SFC). (Bosch Rexroth Corporation 2009) To compare those languages and to estimate their advantages and disadvantages the simple logic is realized on standard IEC 61131 languages. This program consists of four boolean inputs (A, B, C, D) and one boolean output (T). T is true if A is true and B is false or if C is false and D is true. The results are presented in Table 4.2.

The standard IEC 61131 languages became more and more popular. There are many tools for programming PLC using this standard such as CoDeSyS, SIMANTIC STEP 7, Open PCS and many others.

4.5 Conclusion

The EOT crane is a complex object, which consists of many parts and proposes many functions that should be controlled. The modern crane control system is based on the softPLC that manages such devices as trolley, hoist and bridge motors, indication systems and many others. Also this PLC provided a feedback and interactions with the operators. This sort of PLC using provides fast and convenient programming, diagnostics and configuration.

Table 4.2 Standard IEC 61131 languages

	LD	FBD	SFC	IL	ST
Program example				<pre> 0001 LDN B 0002 AND A 0003 ST AB 0004 LDN C 0005 AND D 0006 ST CD 0007 LD AB 0008 OR CD 0009 ST T 0010 0011 </pre>	<pre> PROG1 (FB-ST) END_VAR VAR A: BOOL; B: BOOL; C: BOOL; D: BOOL; T: BOOL; END_VAR IF (A=TRUE AND B=FALSE) OR (C=FALSE AND D=TRUE) THEN T:=TRUE; </pre>
Resemble	graphical language, a series of control circuits	graphical language	graphical language, computer flowcharts	text language, Assembler language	text language, high level computer programming languages
advantages	easy program segmentation; visualization; not many preparation	visualization; easy for user who are not versed in relay	visual nature of the program; suitable for huge program	simple function implementation; movement between different HW platforms; fast execution; less memory requires; more compact; easy to edit and display (compare to graphical languages)	compact program compare to Ladder's program; simple function implementation; movement between different HW platforms; faster execution; less memory; more compact; easy to edit and display

disadvantages	difficult to implement; difficult read of huge program	large amount of screen space; problems with correction of the huge program	requires the large time for program preparing and planning	no visualization, difficult to read, understand, and maintain	no visualization, difficult to read, understand, and to maintain
conversion	FBD, IL, ST	LD, IL, ST	-	LD, FBD, ST	LD, FB, IL
application	simple handling application	simpler programs	application consists of repeatable multi-step process	complex internal functions	complex internal functions
users	service engineers, who are similar with control circuits	users, who are not versed in relay	for developers and users (except internal text functions)	for developers who are similar with assembler, not for service engineers	engineers, who have better background in high level computer languages, not for service engineers

The crane control system requires specified characteristics such as high reliability and rapidity, easily using and service, scalability. Also the system should be smart that means it should prevent operator's mistakes. So, the SW development becomes an important and difficult process and one of its main parts is SW testing that is implemented during different phases of the development process.

Standardization of the cranes, its equipment, functions, and parts permits to automate testing process. So, the set of standard tests to check standard main functions can be created.

5 AUTOMATED TESTING TOOLS

The developers began to create different tools for automated testing as soon as the SW complexity permits to get benefits from their using. In the market the first tools for automated testing appeared in the middle of 1980. Nowadays there are many tools for different SW types. They can be divided into three main groups according to their functions. These groups are test generators, model based input generators and test automation frameworks. (Hartman, Nagin 2004)

The test generators provide the tester with the test inputs and the predicted system outputs, so they perform some of oracles functions. At the same time the test generators do not participate in the test execution.

The model based input generators are aimed to create the input series using information about the SUT and does not produce predicted system output. Such series can be fulfilled manually or with other tools.

The test automation frameworks provide automatically generated or pre-derived test sets and execute the collection without human control.

Choosing of each tools type depends on the tested SW, HW and other tools that are already used. This chapter is focus on the test generators and test automation frameworks choosing, so it includes selection algorithm, main requirements and overview of different tools. This work does not include overview of the model based input generators because they do not produce information to ascertain if a test has passed or failed, so they do not solve the complete test design problem.

5.1 Tools selection algorithm

Choosing of the test automation tools is a project in its own right. (Fewster, Graham 2008) So, it should be founded and resourced adequately. The main steps of this project are shown in Figure 5.1.

During the first step the list of initial requirements should be created. The main characteristics of the tools must be defined. These characteristics include, first of all, compatibility issues, because new tool will not operate alone, but will become a part of the big system. So, the tool should be compatible with the used operation system and development tools.

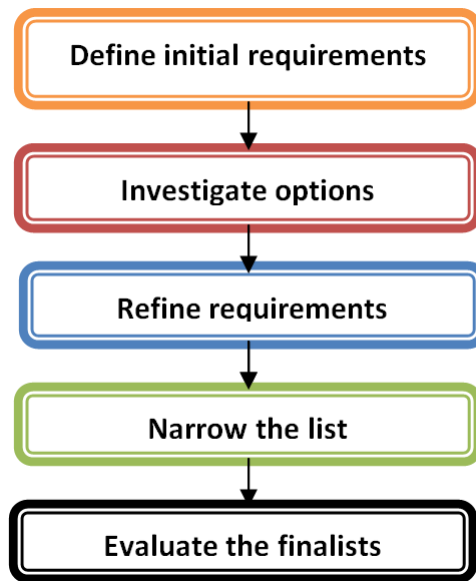


Figure 5.1 Steps of the tools selection process

The benefit that the organization wants to get from the tool should be determined. Also it is necessary to choose a metric of the benefit evaluation.

Then the audience or the tools users should be determined. The skill level of the users should be taken into the consideration. So, for the high level staffers such tools characteristics as flexibility are important. On the other hand, for nontechnical personal ease of use is more important. Thus, the users should be involved in the tools evaluation process. On the other hand, involving too many people with their own private wishes can expand requirements so much that no tool can meet them.

Budget of the testing tools is another significant attribute. The cost has a significant impact on the tools selection. It should be noticed, that the test tools budget comprises of licensing, training, implementation and adding machines costs.

The second step proposes gathering information that includes such actions as Internet combing, newspapers reading, conferences attending, advertisements checking out and

many other activities the main goal of which is investigation of the different tools. The cooperation with tool sales people is inevitable action of this step.

During the second step, the new additional requirements that should be put in the requirements list can be discovered. Also the new information about tools allows making deep investigation about users' requests. So, the next step permits the requirements list correction, adding and refining.

As usual the list of requirements can be quite big and no one of the tools meets all points. At the same time several tools satisfy some of them. To determine the tools that is more suitable the requirements should be prioritized. The action should be done during fourth step to save time, because the evaluation is very difficult process. Different tool can have some related characteristics, thus not all points must be prioritized. Also if it turns out that there is only one suitable tool on the market this step is not needed at all. So, the evaluation should be done in the end of the tools choosing process. Moreover, the fourth step includes the tools selection and purchasing.

The last step intends the evaluation of the finalists. So, one or more tests should be created, checked and run using each of tools. Also the tool is checked in the case of user interface changing to estimate needed efforts for the test adaptation. The number of variation deepens on the plenty of time. Then the result evaluation is done and tools are compared and one of them is chosen according to the results of this comparison.

After the needed tool is found, the huge work of its implementation is started. This part consists of main activities such as training organization, measures calculation and many others.

5.2 Test generators

Most of the test generators are based on the SUT models. (Cherukuri, Gupta 2010) Models implementation allows considering the SW at the abstract level and reducing its complexity. So, model based testing (MBT) is used to generate efficient test cases in a short period of time. This approach can use different models such as domain, environment,

behavior, and abstract model. The test generators use as usual the behavior model because it permits to involve the oracle information and create the expected output values of the SUT. (Utting, Legeard 2007)

MBT is shown in Figure 5.2 and includes five main steps. During the first step the SUT or its environment model is created. The model must be small compare to the system size, otherwise the product cost increases. On the other hand, it must describe accurately the needed characteristics and the key aspects, so it should be detailed enough. After the model writing it should be proved that this model is consistent and has specified behavior.

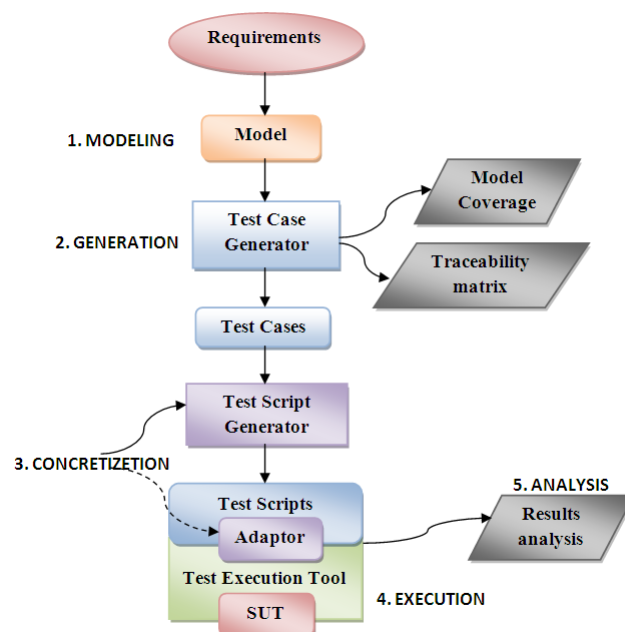


Figure 5.2 The MBT concept. Adopted from (Utting, Legeard 2007)

The next step means abstract tests generation from the model. Also it proposes the test criteria selection to define needed test from the infinite number of possible tests. The first two steps distinguish MBT from other kinds of SW testing.

Then the concretization of these tests should be done to transform them from abstract to the executable concentrate tests. The last two steps consist of tests execution and result analysis.

MBT is widespread because of its benefits. First of all, it is indeed good at finding the SUT defects. (Utting, Legeard 2007) Also for many cases the time that is required for model creation and maintaining and test generation is less than the time that is spent for the

manual test cases design. So, MBT leads to less testing time and effort. Moreover, the test cases that are generated based on MBT do not depend on testers' ingenuity. Therefore, the test quality is improved.

At the same time MBT has limits. MBT proposes models that are written manually, so it depends on the skill and experience of the people who create the models and chose the test criteria. Some of the SUT parts can be difficult to model, so it can be quicker to test them manually. Moreover, test fail can be caused not only by the SUT defects, but by the model errors as well.

So, MBT is a powerful approach for automated testing, but has the week point that is the SUT model.

5.2.1 Initial requirements

The main purpose of the test generator is producing the test cases according to the different criteria. At the same time it should provide the test cases analysis, convenient tools for behavior modeling, and some other activities. So, the test case generation tools should:

- have low costs;
- generate tests according to the different test criteria;
- have skills to build a plant model or a finite-state machine (FSM);
- have visualization;
- be able for result analysis;
- create the etalon outputs

This section represents the main generator test requirements in general form. They are specified and prioritized in next sections.

5.2.2 Investigate options

There are many different test generators at the market but most of them were created to generate the test cases for web applications. This section presents snapshots of the different

generators that are used to create the test cases in the embedded systems and have a good reputation. Another criterion that was considered during option investigation is prevalence. So, six tools that satisfy those demands were revealed.

Conformiq Tool Suite is a commercial true MBT generator that consists of *Conformiq Modeler* that is a tool for SUT behavior modeling using UML (Unified Modeling Language) and QML (based on Java Script) and *Conformiq Designer* that is a tool for optimizing test design. It generates a high-quality set of the test cases mathematically from the high-level system model and exports the test cases in user-definable formats, for example TCL, TTCN-3, Visual Basic, HTML, XML, or Python. (Conformiq) Conformiq tools are used for wide variety of systems.

ModelJUnit is one of the most famous open MBT generators that is based on Java. It generates test sequences from the Java FSM/EFSM models and values different model coverage metrics.

Reactis Tester proposes automatic test generation from Simulink models. Also it provides comprehensive coverage of different test coverage metrics (one of several offered metrics can be chosen). Tests are viewed as a matrix that has rows, which represent inports and outputs, and columns, which correspond to simulation steps.

TGV (Test Generation with Verification technology) is a tool for the automated test generation that requires a protocol's behavior and a test purpose. The behavior's description should be given as an Input/Output Labelled Transition System (IOLTS). The test cases are generated in the standard TTCN format.

Tigris is another open test generator that is written in Java and generates tests in a similar way to ModelJUnit. It requires the models, which are drawn graphically using any graphml-compatible editor, but supports no its own modeling. It is a good generator for simple models.

TVEC has a Test Vector Generation System that needs the models of the requirements and the behavior of the system in a proprietary language called T-VEC Linear Form and have no support for UML or other widely used modeling languages. Also it permits to use SCR models and automatically translating them into the Linear Form. The test cases are

generated based on the domain testing theory. So, TVEC generates the test cases by parsing the logical expressions in the specification model and defining the extreme values.

5.2.3 *Refine requirements*

During the options investigation it became obviously that requirements should be refined and specified because most of them are satisfied by all examined test generators. So, they can be toughened to evaluate different tools and choose the best one.

The test cases can be generated according to many test criteria, but they do not have the same efficiency. The transition coverage is one of the best criteria, but it is necessary to have at least two criteria. Another good and simple criterion is random, which usually creates test cases that define many defects.

The most of the test generators have special tools for modeling the SUT behavior. So, the requirement should be specified, to reduce train and tools cost, the model should be done using widespread languages like UML, Java or Simulink.

Different test automation frameworks use inputs in different forms, so it is needed to have test cases outputs in different standard forms for easy integration. Also this form should be convenient for testers that are going to work with outputs during, for example, results analysis.

5.2.4 *Narrow the list*

Table 5.1 represents how the tools satisfy the main requirements. Green box points requirements that are fully satisfied by the tool, orange singles out requirements that are satisfied but not to the full, red means the requirement is not satisfied at all.

No one tool satisfies all requirements, but there are two leaders (Conformiq and ModelJUnit), which should be examined in the greater details.

Table 5.1 Results of tools evaluation

Requirements Tool name	Cost	Test criteria	Tools for model	Model language	Outputs format	Visual ization	Result Analysis	Oracle information	Cite
Conformiq	comm	State Coverage, Transition Coverage, 2-Transition Coverage, Implicit Consumption and many other	yes	UML state diagrams, java script	HTML, TTCN-3, Perl, TCL	yes	yes	yes	http://www.conformiq.com/
modelJunit	free	Random, Greedy, Look Ahead Walk	no	Java	Self written	yes	yes	yes	http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/
Reactis Tester	comm	random	no	Simulink models	Matlab formats	yes	yes	yes	http://www.reactive-systems.com/
TGV (Test Generation with Verification)	free	depth-first search algorithm	no	LOTOS, SDL, or IF specification model.	TTCN	no	no	yes	http://www-verimag.imag.fr/~async/TGV/
Tigris	free	A_STAR, RANDOM SHORTEST_NON_OPTIMIZED	no	FSM, flow graph (Yed)	Java, Perl	yes	no	yes	http://mbt.tigris.org/
TVEC	comm	reach extreme, values in the decision path	no	T-VEC Linear Form, SCR models, not UML	Java, Perl, C++, Python	no	yes	yes	http://www.t-vec.com/

5.2.5 Evaluate the finalists

The SUT is needed to examine two finalists (Conformiq and ModelJUnit) deeper. The logical part of the PLC SW that is used to provide power supply in the crane is chosen. Crane On control supplies power to machinery drives and motors by energizing the main contactor. The flow graph of the SUT behavior is presented in Figure 5.3.

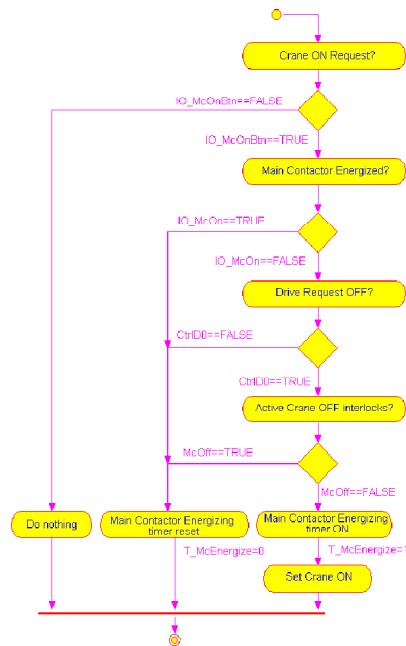


Figure 5.3 Crane ON control logic

Crane on sequence is started when:

- crane On-request is ON from the active control place
- main contactor circuit feedback must be off
- all machinery direction requests from selected control place are off
- no active Crane OFF interlocks

Modeling

Conformiq Tool Suite has special tools *Conformiq Modeler* to create the SUT behavior model. It permits to use UML graphical language or QML. The Crane On logic model that is realized using UML is presented in Figure 5.4.

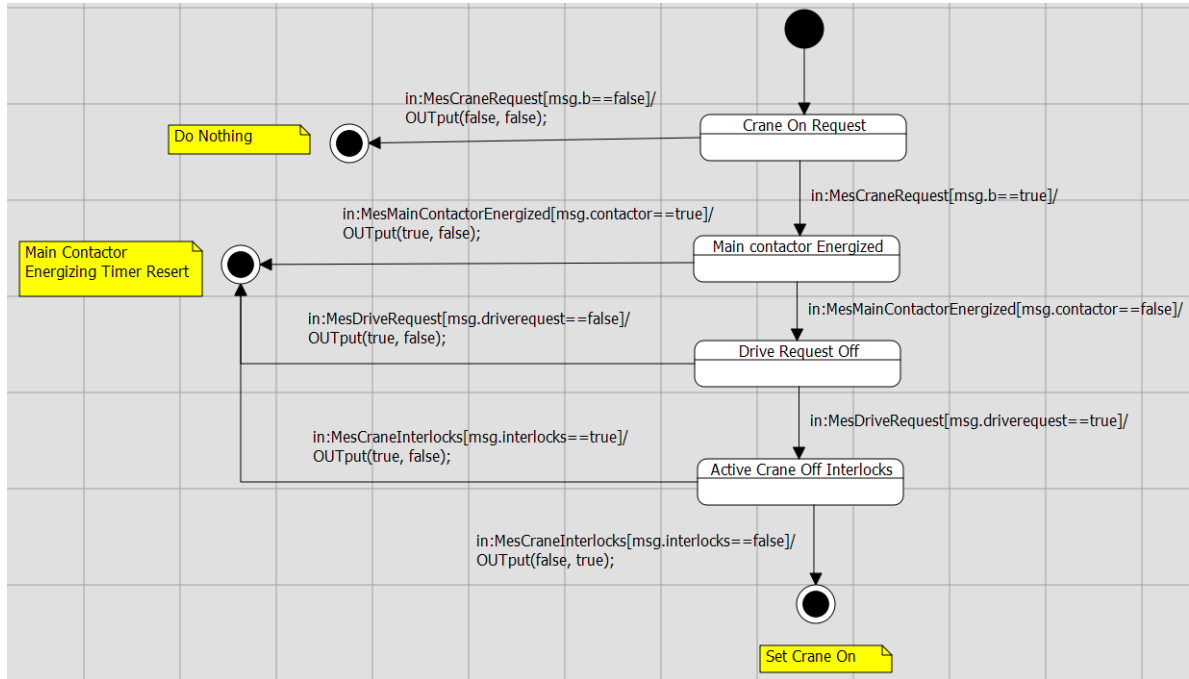


Figure 5.4 Model of Crane On logic (Conformiq, UML)

The description of outputs and inputs, interlocks between them is written in specified file using QML (appendix A).

ModelJUnit does not have any tools for modeling, but it requires a behavior model written on Java. So, it can be done in Notepad (appendix B) and saved with extension *.java*. It consists of two parts: model of the SUT and test characteristics by default that can be changed later. This java file should be compiled, so the file *.class* is received.

Execution

Conformiq allows test cases generation for different testing goals that Conformiq Designer tries to cover. These goals are changed in Cover Editor (Figure 5.5). So, Conformiq Designer proposes coverage of states, transitions, 2-transitions, conditional branches, and control flow. Also, it provided different set for values such as “Target” (Conformiq Designer tries to generate at least one test case that will cover this goal), “Block” (Conformiq Designer avoids generating tests that cover such goal), and “Don’t care” (it can be covered but if it is not, it will not affect the total coverage information).

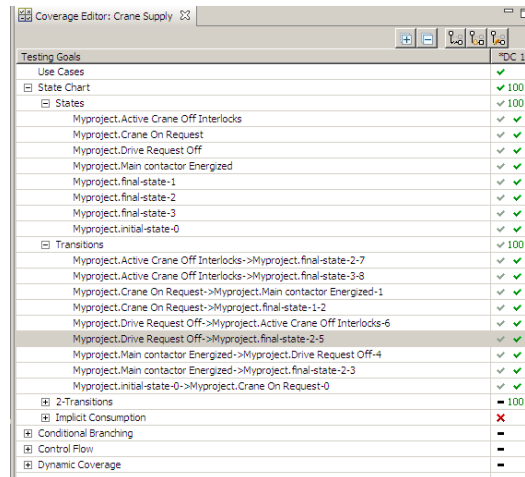


Figure 5.5 Conformiq Designer Cover Editor

The characteristics of the test generation in *ModelJUnit* can be changes using Test Configuration window (Figure 5.6). First of all, the test generation algorithm should be chosen. ModelJUnit proposes such algorithms as Random walk, Greedy walk and Lookahead walk. Also, the total test length can be designated.

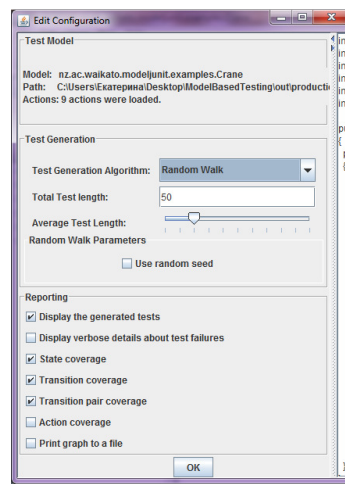


Figure 5.6 ModelJUnit Test Configuration window

Result analysis

Conformiq permits to present generated test cases using different forms. The Test Case List (Figure 5.7) has the list of all generated test cases numbered from first to last. The test cases that are generated under current conditions are black, red are tests that were created earlier under previous targets.

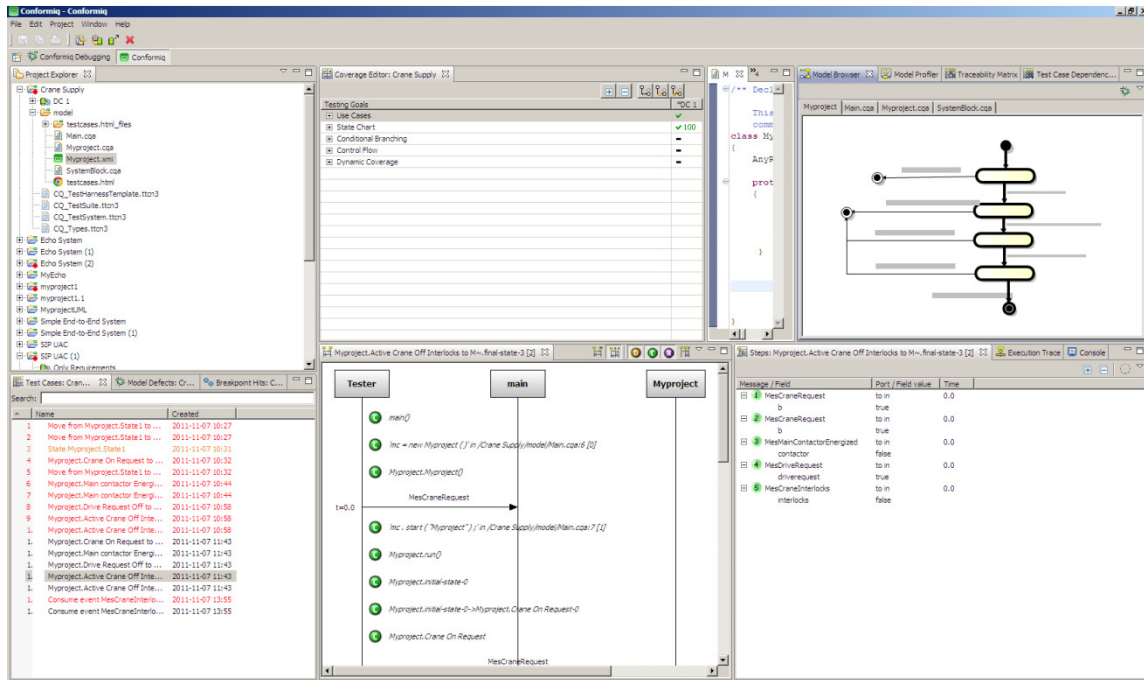
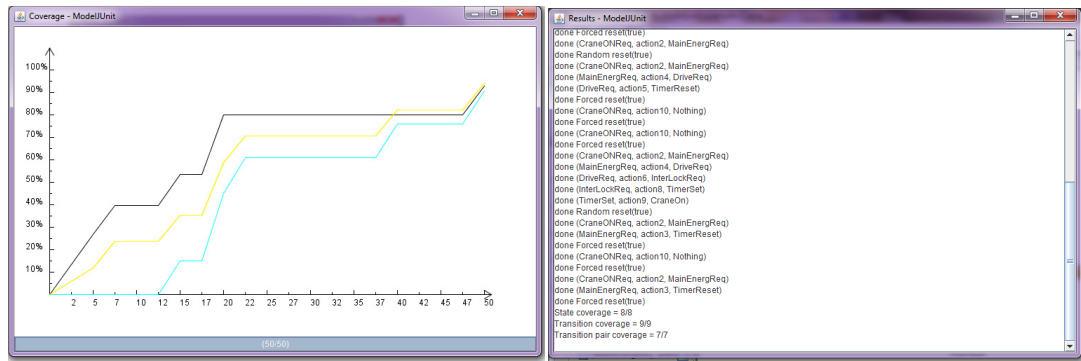


Figure 5.7 Conformiq output viewing

Message Sequence Chart (MSC) can be seen in the Test Cases window (Figure 5.7). An arrow that goes from the 'Tester' to the 'Myproject' means that the test environment sends an input to the system and an arrow that leaves the 'Myproject' means that the system has sent out.

Also, Conformiq provides test cases overviewed using Traceability Matrix and Test Steps. The Traceability Matrix represents the testing goals linked with the test case number that covers this goal. The payload data that are used in the active test are presented in the Test Steps window (Figure 5.7).

ModelJUnit proposes different reports. The report properties are defined in Test Configuration window (Figure 5.6). Report can include generated tests, test failures, state, transition, action, and transition pair coverage. Also, this characteristic can be presented in a graph (Figure 5.8). One report is presented in Figure 5.8 and includes created test cases and different metrics.



(a) (b)

Figure 5.8 ModelJUnit Result analysis: a) graph; b) results report

The graphs on the Figure 5.8 represent State (black), Transition (yellow) and Transition Pair (blue) coverage changing during test cases creation.

ModelJUnit main window permits to determine which states and transitions are covered by each test step (Figure 5.9). The test step is chosen in the left part of the window. The covered graph is marked out in the top right part and is shown separately in bottom part of the window.

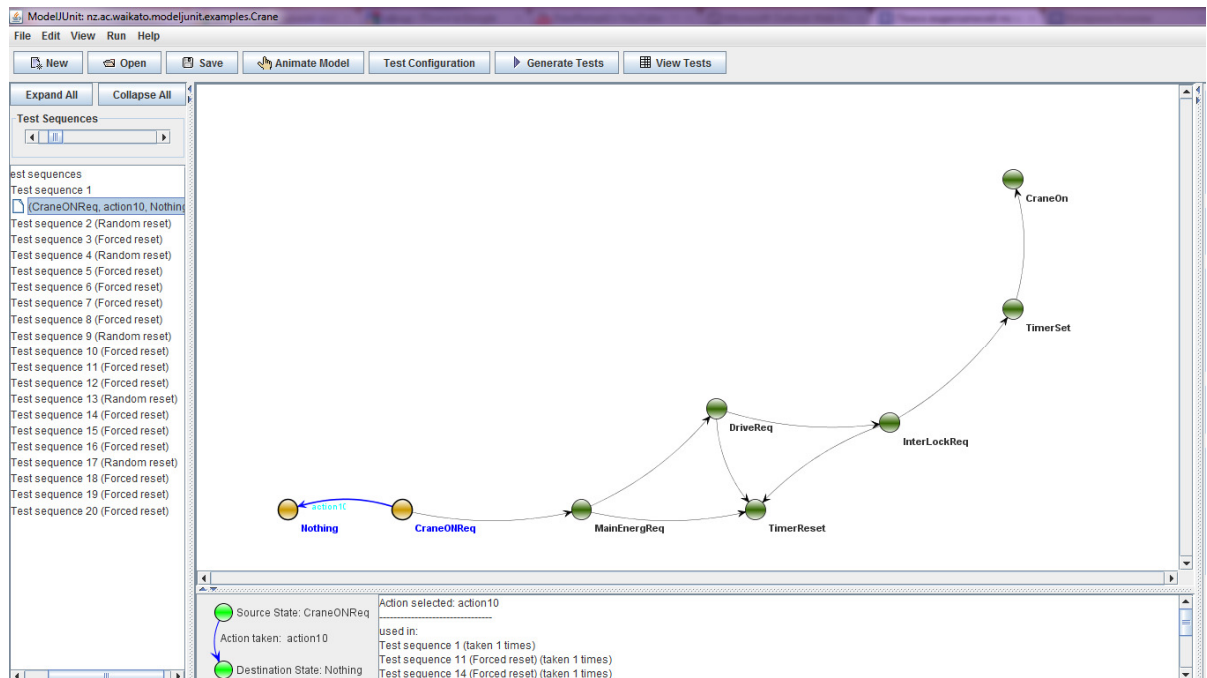


Figure 5.9 The ModelJUnit Main Window

5.2.6 Conclusion

Conformiq has special tools for the SUT model creation that are based on UML and QML (Figure 5.4). QML is based on Java Script and can be used by programmers, who are not familiar with UML language. Also Conformic proposes easy visualization and understanding of the model by using different comments (yellow in Figure 5.4). On the other hand, ModelJUnit uses Java language for the model creation that is more convenient for programmers and proposes model visualization during the test cases creation (Figure 5.9). This model visualization is created by the program, so it requires more time for programmer to check it compare to the Conformiq model (Figure 5.7) that is made by the programmer, so ModelJUnit model permits to discover logic defects. It is difficult to say which of tools is better; both of them have advantages.

Conformic proposes different test criteria that can be used, for example the test cases can be generated to cover different states, paths or even part of parts. So, if only one state should be tested by any path only one test case will be created. At the same time ModelJUnit proposes only three test generation algorithms and no target algorithm. So in the same case, many test cases are going to be created and the tester should chose suitable one. So, Conformiq is more convenient.

Conformiq offers many different ways of generated test cases visualization and analysis (Figure 5.7). So, there is a simple test cases list, separated presentation of test cases steps, and graphical representation of them on the SUT model. Also the test coverage is shown in the coverage editor. Traceability Matrix permits to evaluate coverage of each test case. ModelJUnit presents the model of SUT and covered and uncovered states. (Figure 5.9) Also it generates the report that includes generated test sequences and the test coverage (Figure 5.9, Figure 5.8) and graph of the test coverage can be drawn. Therefore, Conformiq has more possibilities for generated test cases, but ModelJUnit has enough skills for it as well.

Conformig permits to export the test cases in user-definable formats, for example TCL, TTCN-3, Visual Basic, HTML, XML, or Python. *ModelJUnit* proposes self written output. These test cases easy to analyze for the tester and then they can be transformed in any

needed form using different tools. It is difficult to say, which of tools has preference because test necessary outputs format depends on used test framework.

ModelJUnit and Conformiq have many advantages. One of the main Conformiq advantages is possibility of using different target criteria and different output formats. The main Conformig disadvantages is its price, at the same time ModelJUnit is open source and has enough skills to generated good test cases and analyze it. Therefore, the final conclusion of the test generation tools will be done after the test framework choosing.

5.3 Test framework

Automated test framework is a tool for automated test cases execution. Hence it provides some other functions during testing process.

Test execution workflow consists of several steps (Figure 5.10). (IBM) After the creating of the test case it should be add to the test plan. The test plan is needed to organize test execution activities. The next step proposes test environments definition. Then a test script is created and associated with the test case. The created test script is run on the SUT. The test execution results are putted in the final report. Analysis of the testing results is made to indentify if the SW passed the test.

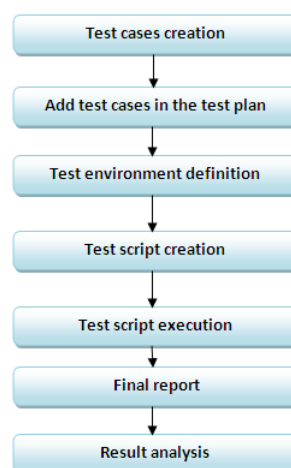


Figure 5.10 Test execution workflow

A good test framework is responsible for the automated running the test scripts on the SUT, comparison the gotten results with the etalon outputs, and creating final report. It does not take part in the result analysis, because this step is very difficult and, as usual, is made manually.

5.3.1 Initial requirements

The main purpose of the test framework is automated execution of the test cases that were generated before. At the same time it should provide test result analysis and final report generation. So, the test execution tools should:

- use standard languages for test scripts
- be used for embedded systems
- have possibility to interface with PLC and field bus;
- propose real-time execution
- consist a test evaluation part that compares expected outputs with actual outputs
- use different comparisons algorithms
- produce a test report
- have low cost

5.3.2 Investigate options

This section presents snapshots of main frameworks that can be found in the market.

TestPartner is a tool for automating functional testing for .NET, Web, and a wide range of applications based on heterogeneous technologies.

Mercury WinRunner is automation functional testing tool for Microsoft Windows applications.

National Instruments (NI) presents family of tools for automated testing including SW and HW. NI VeriStand and TestStand are software tools for performing real-time testing applications more efficiently. Also NI offers different IO models.

Rational Robot is an automation tool for testing client/server applications.

The *Reactis* tool suite of Reactive Systems, Inc. proposes automated test execution based on Simulink/Stateflow control system models.

TAU Generation2 is a family of tools that are used to simplify, automate and accelerate the development of advanced systems and software based on SILS.

5.3.3 Refine requirements

There are many tools for automated test execution for software running on a desktop computer, but only few for embedded software. As usual, the desktop software uses a keyboard, a mouse, a database or a file to get input and a monitor, a printer, a file or a database to display outputs, which are standardized. So it is easy to organize automated tests execution. The most widespread problems of automated test frameworks for the embedded system software are inputting data and reading responses from the SUT.

Another huge difference between the desktop computers SW automated testing process and SW automated testing in embedded systems is the interface to the system. The regular interfaces are provided by the operational system. Interface of the embedded system depends on the hardware, and special software that simulates the devices that sends and receives signals to and from the embedded system software should be developed to interact with the system over the interfaces.

These specified characteristics of the embedded system are reflected in the first four requirements from the section 5.3.1. Therefore, it should be noticed that these requirements are the most important and should be satisfied first of all.

5.3.4 Narrow the list

Table 5.2 shows how the mentioned earlier frameworks satisfy the requirements.

There is only one tool (National Instruments tools), that satisfies almost all requirements. It is necessary to evaluate this tool to make sure it is suitable for the discussed system.

Table 5.2 Results of the test framework tools evaluation

	be used for embedded systems	can be used with the PLC	can be used with different buses	use standard languages for test scripts	consist a test evaluation part	use different comparison algorithms	real-time execution	produce test report	low cost
Compuware TestPartner	no	no	no	x	no	no	no	x	no
Mercury WinRunner	no	no	no	x	x	no	no	x	no
National Instruments tools	x	x	x	x	x	no	x	x	no
Rational Robot	no	no	no	x	x	x	x	x	no
Reactis Simulator and Validator	x	no	no	x	x	x	no	x	no
Robotframework	no	can be	can be	x	x	no	no	x	x
Telelogic's Tau Tester	x	no	no	x	no	no	x	x	no

5.3.5 Evaluate the finalists

NI offers family of the tools for automated test cases execution that includes SW and HW.

NI TestStand is a test executive to automate the execution of code modules written in any programming language. Also this software generates different test reports. NI TestStand steps are not based on any increment of time, so sequences cannot be run with any measure of determinism.

The simplest scheme for the system SW testing are shown in Figure 5.11 and consists of PC, which has NI TestStand for test execution and test manage, buses drivers and card to send and receive signals to and from the PLC and PLC with software that is tested. The PLC software can be change using the second PC with installed SW development environment for PLC applications. The connection between the second PC and PLC is made using Ethernet.

The advantages of this system are its simplicity and low cost. At the same time NI TestStand is Windows-based software that does not run on any real-time operating system, so the scheme is not used for real-time testing.

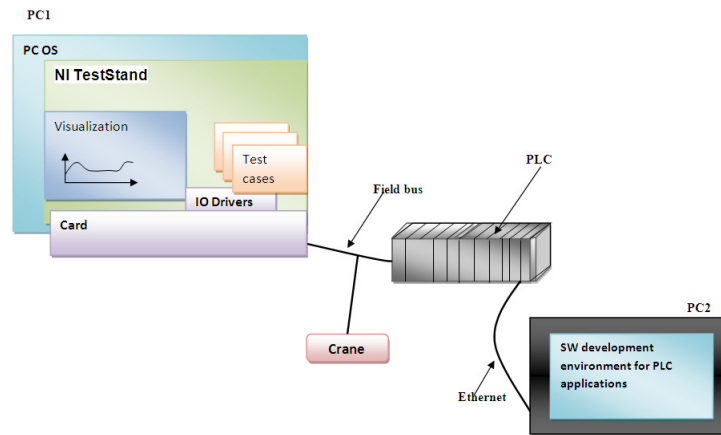


Figure 5.11 Scheme of automated SW tests execution based on NI tools

To execute real-time tests a real-time application environment such as NI VeriStand that is used with PXI should be implemented (Figure 5.12). NI VeriStand is real-time testing software designed to perform real-time validation of embedded control systems and physical test cells. (NI 2011) PXI is a rugged PC-based platform for measurement and automation systems. (NI 2010)

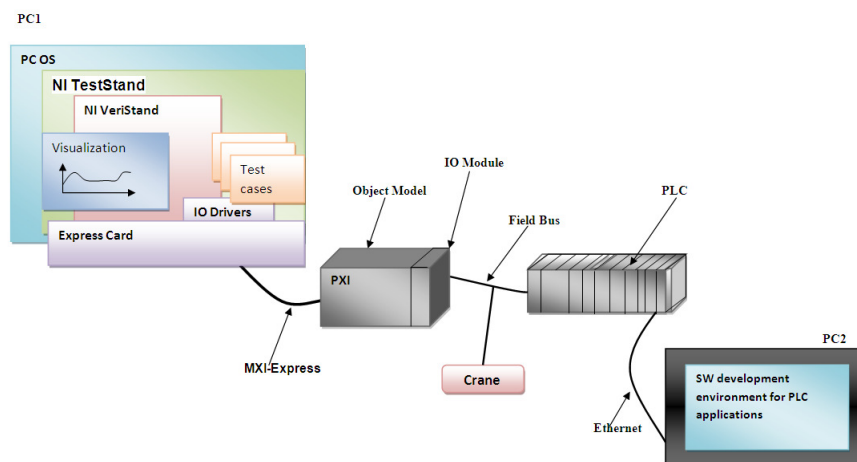


Figure 5.12 Scheme of the real-time automated SW tests execution based on NI tools

One of the main disadvantages of this system are complexity and high cost. At the same time such system permits using hardware-in-the-loop simulation (HILS) during testing. Simulation is one of the most widespread techniques that are used to test different control systems. The simulation can be defined as the art to make things that are not exist yet working together. (Kramer 2001) So, to increase the effectiveness of the design and the testing process the integrative control and model system development is implemented.

HILS allows testing the developing control SW subsystems under the real-time conditions (Figure 5.13). It requires control SW porting to the PLC. So, the PLC system plays the HW component in the loop. At the same time the plant simulation model is transferred to an appropriate real-time simulation platform. NI offer special tools LabVIEW for the model creation.

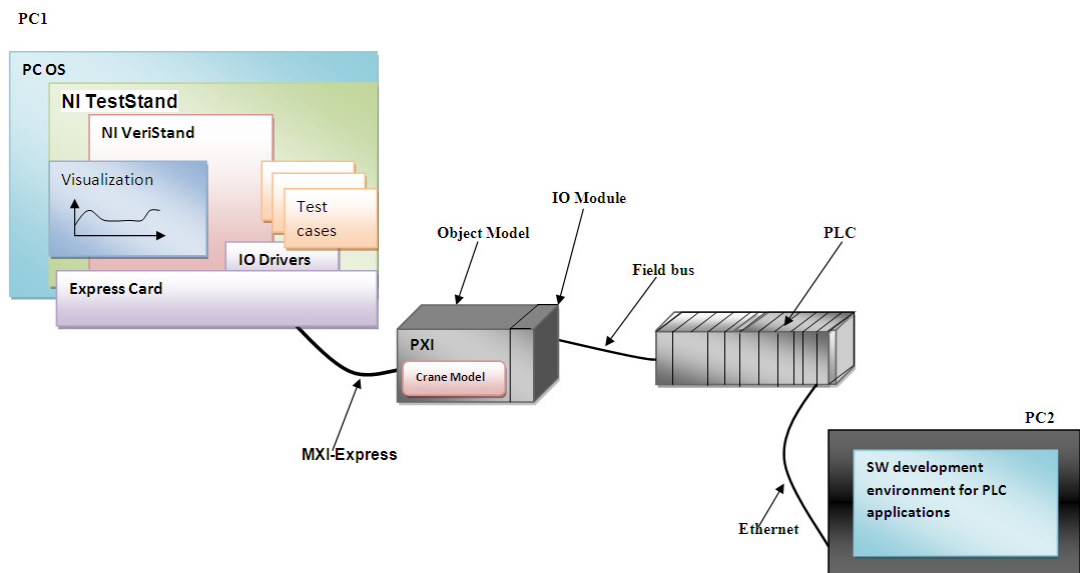


Figure 5.13 HIL testing

However, this approach has huge disadvantages: it proposes the plan simulation, which requires much time and independent development, because the simulation should not be the tester's process interpretation.

One of advantages of using plant emulation during the PLC SW testing is facilities of immediately trying the logic sequence as soon as it is done. So, the immediate feedback is achieved. It is useful in the many cases. First of all, during the development of a new product its simulation allows to start the SW development and checking before the final part is ready. It provides the parallel the product and the SW development and reduces the time costs. On the other hand, the plant simulation is useful, if the project is far from the SW developers and it takes much time to go to the plant and check the SW. So, the simulation permits decreasing the testing time. Also the plant simulation allows checking the data flow and the analog values and the test operator interface functions. Therefore, word type instruction is checked. The problem of testing safety is solved by the simulation as well. Also using plant simulation instead of the real plant reduces the tester's stress.

Automated tests execution is used not only for the system testing, but for other types of tests as well. Figure 5.14 presents a scheme that is good for integrated tests. On the one hand, the system is simple and inexpensive. At the same time the connection between the PC and PLC is done using Ethernet, so it is impossible to make some changing in the SUT during testing.

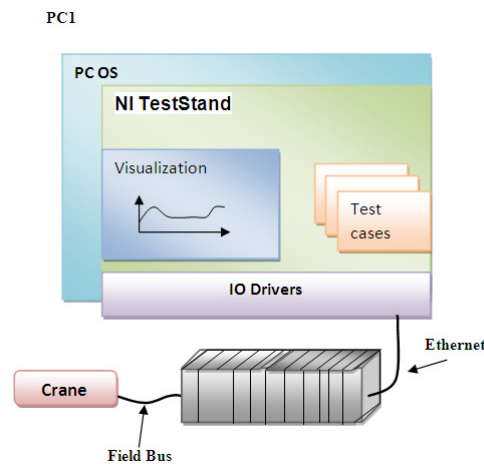


Figure 5.14 Integrated testing

5.3.6 Conclusion

National Instrument tools were designed to develop and test embedded systems. The automated testing system that is build using these tools comprises SW and HW parts and satisfies almost all specified requirements. First of all, NI provides HW and drivers for communication PLC. Also NI TestStand is SW for automated execution of tests, which are written on the any standard language, and different final reports generation. Special tools such as NI VeriStand and PXI are used for real-time tests execution. Another advantage of the system is possibility of HIL testing.

The main disadvantages of this system are complexity and high cost. The cost includes not only the cost of necessary HW and SW, but cost of system implementation, service and staff training as well.

5.4 Conclusion

In the beginning of this chapter the algorithm for choosing the automated testing tools is represented. According to this algorithm tools for automated test cases generation and execution are selected. The automated test cases generators are based on MBT that allows creation of the test inputs and test etalon outputs. One of them is Conformiq that has special tools for modeling the SUT, creating test cases under different criteria and exporting test cases in different forms such as HTML, Java, C#. It makes Conformiq the leader in the automated test cases generator's market.

Automated test framework is responsible for the automated running the test scripts on the SUT, comparison the gotten results with the etalon outputs, and creating final report. Automated test execution for embedded systems proposes inputting data and reading responses from the SW. Among examined automated test frameworks only NI offers tools for creation of automated testing system that satisfied specified requirements. The system includes SW (NI TestStand, NI VeriStand, NI LabVIEW, different drivers) and HW (IO models, cards, PXI) and proposes real-time test execution and HIL SW testing. The main disadvantages of the system are its complexity and cost.

6 DISCUSSION AND CONCLUSION

6.1 Results of the work

The test automation process consists of two parts: tools selection and implementation. The present work presents tools selection for automated embedded system SW testing. In this thesis tools for automated test cases generation and execution were selected. Therefore, the testing system comprises of several parts and is presented in Figure 6.1.

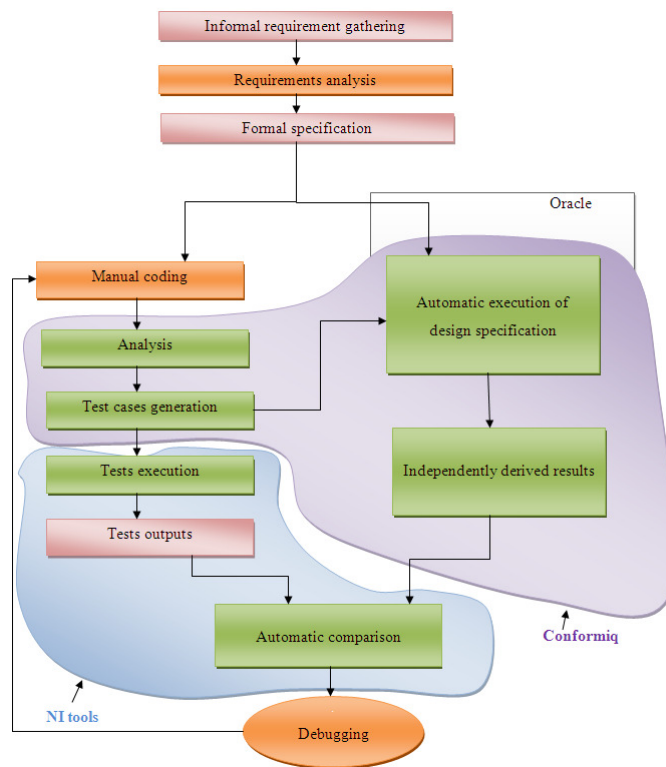


Figure 6.1 Automated SW testing

First of all, according to the product specification the software is coding. At the same time according to the same specification the SUT model is done in Conformiq Modeler by developers. This tool proposes the model creation using Java and UML languages. Then the criterion that is going to be used to create test cases is chosen. Next step proposes automated test cases and expected outputs generation based on the analysis of the SUT model by Conformiq Designer. After this step the analysis of created test cases should be done. If the test coverage is high enough, the test cases are exporting in convenient forms (HTML for testers and Java for future using).

After test cases generation they should be executed. NI TestStand and NI VeriStand are used for automated real-time test cases execution. The execution system includes not only SW but needed HW (PXI and IO models) as well. After test execution NI TestStand compares gotten outputs and expected outputs and makes the final report. The final step of testing process that is result analysis is done manually by testers.

The main advantage of the system is possibility of different tests execution such as integrated, system and HIL tests. So, this system can be used during different design steps.

Automated testing system reduces testing time and permits automated test execution in the night time. Also the system has convenient tools to determine test coverage and generate different reports.

The test cases that are produced by the present system are reputable and reliable.

The test execution is based on the tool from one company (NI), so there should not be many problems with its implementation.

The main disadvantages of the system are its cost and complexity. The cost includes cost of necessary SW and HW, the cost of their service, implementation, and cost of staff training.

Another weak point of the system is SUT model creation. First of all, it takes time and human resources. On the other hand, the time for test generation is reduced. So, in the case of regression testing that proposes little model changes and requires new test cases the time needed for SUT model creation and automated test generation is less than time needed for manual test cases generation. Also, the model defects could cause test faults, so the test result analysis becomes more complex. At the same time the model and SUT code are done according to the one specification by different people. Such different views on one specification help to identify its defects.

The present system does not provide full automated testing process and proposes human activities during different test steps such as SUT model creation.

Successful tool selection is no guarantee of successful tool implementation within the organization: 40-50% of organizations have test tools that are 'shelfware'.

6.2 Future work

The testing automation process is shown in Figure 6.2. This work presents the first part and includes targets of automated testing, testing tools requirements, their investigation, evaluation and selection. Selection of the automated testing tools is only the first step towards the test automation.

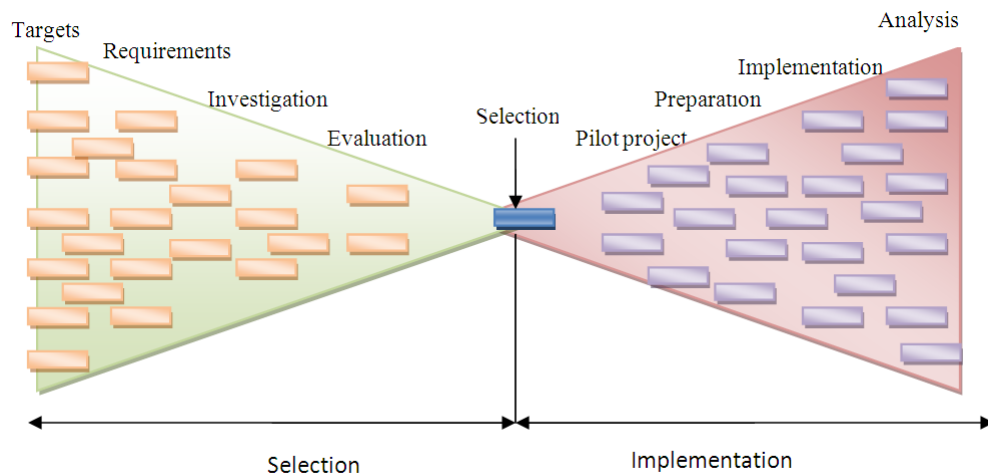


Figure 6.2 The automation process

After selection the implementation part that is a process of gradually widening the tool's acceptance, use and benefits starts. (Fewster, Graham 2008) It is better to start this phase using pilot project that allows experimenting and discovering new testing system. "If you don't know what you're doing, don't do it on a large scale". (Fewster, Graham 2008) The pilot project shows main problems and benefits of the automated testing. Unfortunately, there was not enough time for the pilot project during this work.

If the pilot project is success, tools can be safely used for huge projects. If the pilot project was failed, then the tools are not suitable or were used in not suitable way.

Implementation of new system always has effects that were not expected. Preparation part includes exposing how the tool will affect the testing process and how this process should be modified. This phase includes dividing the test cases into automatable and no and definition of the testing levels. Also many organizational issues are consumed during this step.

When all preparation problems are solved, the widespread tools using begins. The development team should be ready to force new problems and issues during this phase. For example, it is difficult to persuade people to change the way they work.

Last step proposes analysis of the testing system and estimation of the benefits. Automated testing is the process that should be always under development and modification. So it is never-ending process.

The implementation part of the automation testing provides a set of issues for further research.

REFERENCES

- Anonymous a, *Dictionary*, Available: <http://dictionary.reference.com/> [cited 05.05.11].
- Anonymous b, *Technical Translation Dictionary & Glossary: English Portuguese Dutch*. Available: <http://technical-portuguese.blogspot.com/> [cited on 05.05.11].
- Abrahamsson, P., Salo, O., Ronkainen, J. & Warsta, J. 2002, *Agile software development methods. Review and analysis*, VTT publications, Finland, Espoo.
- Adrion, W.R., Branstad, M.A. & Cherniavsky, J.C. 1982, "Guideline for Lifecycle Validation, Verification, and Testing of Computer Software", *Computing Surveys*, vol. 14.
- Adrion, W.R., Branstaad, M.A. & Cherniavsky, J.C. 1982, "Validation, verification, and testing of computer software", *ACM Computing Surveys (CSUR)*, vol. 14.
- Bach, J. 1996, "Test Automation Snake Oil", *Windows Tech Journal*, vol. 10.
- Baker, C. 1957, *Review of D.D. McCracken's "Digital Computer Programming"*, *Mathematical Tables and Other Aids to Computation* 11.
- Bartels, G., Kist, P., Schot, K. & Sim, M. 1994, "Flow Management Requirements of a Test Harness for Testing the Reliability of an Electronic CAD System", *EUROASIC, the European Event in ASIC Design* IEEE, France, Paris.
- Beck, K. 1999, "Embracing Change With Extreme Programming", *Computer*, vol. 32.
- Beizer, B. 1990., *Software testing techniques* 2th edn, Boston (MA) :, International Thomson Computer Press.
- Bertolino, A., Gao, J., Marchetti, E. & Polini, A. 2007, "Automatic Test Data Generation for XML Schema-Based Partitioning Testing", *Second International Workshop on Automation of Software Test*.
- Bicego, A., Jacobone, J., Maiocchi, M. & Poggi, U. 1986, "Towards automation in software quality control: The case of products described by formalgrammars ", *IFIP Congress*.
- Bosch Rexroth Corporation 2009, "Understanding the IEC61131-3 Programming Languages" .
- Bourque, P. & Dupuis, R. 2004, *IEEE Guide to Software Engineering Body of Knowledge*, Angela Burgess, USA.
- Brown, D., Roggio, R. & McCreary, C. 1992, "An Automated Oracle for Software Testing", *IEEE Transactions on Software Engineering*, vol. 41.

- Cherukuri, V. & Gupta, P. 2010, *Model-Based Testing for Non-Functional Requirements*, Mälardalen University.
- Ciupa, I., Leitner, A., Oriol, M. & Meyer, B. 2008, "ARTOO: Adaptive Random Testing for Object-Oriented Software", *30th International Conference on Software Engineering*.
- Conformiq , *Automated Test Design* . Available: <http://www.conformiq.com/> [cited on 18.07.2011].
- Cornett, S. 2011, "Code Coverage Analysis", *Bullseye Testing Technology*, [Online], Available from: <http://www.bullseye.com/>.
- Coulter, A.C. 2000, "Graybox Software Testing in Real-Time in the Real World", *Software Quality Engineering*.
- DeMillo, R.A. & Offutt, A.J. 1991, "Constraint-Based Automatic Test Data Generation", *IEEE Transactions on Software Engineering*, vol. 17.
- Dhillon, B.S. 1999, *Design Reliability - Fundamentals and Applications*, CRC Press, USA.
- Duran, J.W. & Ntafos, S.C. 1984, "An Evaluation of Random Testing", *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4.
- Fewster, M. & Graham, D. 2008, *Software Test Automation Effective use of test execution tools*, Idea Group Publishing, New York.
- Garrett, T. 2009, "Implementing Automated Software Testing - Continuously Track Progress and Adjust Accordingly", *Methods & Tools*, [Online], vol. 17, no. 3. Available from: <http://www.methodsandtools.com/>. [cited on 05.07.2011].
- Garrett, T. 2011 "Useful Automated Software Testing Metrics", *Software Testing Geek*, [Online], Available from: <http://www.softwaretestinggeek.com/>. [cited on 08.07.2011].
- Gelperin, D. & Hetzel, B. 1988, "The Growth of Software Testing", *Communications of the ACM*, vol. 31, no. 6.
- Goodenough J.B., G.S.L. 1975, "Toward a Theory of Test Data Selection", *International Conference on Reliable Software* ACM, USA, New York.
- Graves, T.L., Harrold, M.J., Kim, J., Porter, A. & Rothermel, G. 1998, "An empirical study of regression test selection techniques", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 10, no. 2.
- Hartman, A. & Nagin, K. 2004, "AGEDIS tools for model based testing", *International Symposium on Software Testing and Analysis* ACM, USA, New York.
- Hartmann, J.R., D.J. 1990, "Techniques for selective revalidation", *Software, IEEE*, vol. 7, no. 1.

- Hong Zhu, Hall, P.A.V. & May, J.H.R. 1997, "Software Unit Test Coverage and Adequacy", *ACM Computing Surveys (CSUR)*, vol. 29, no. 4.
- IBM, *Test execution overview*. Available: <http://www.ibm.com> [cited on 25.08.2011, .
- IEEE Std 829-1998 1998, *IEEE Standard for Software Test Documentation*, New York.
- IEEE Std.610-12 1990, *IEEE Glossary of Software Engineering Terminology*, ANSI.
- John, K. & Tiegelkamp, M. 2010, *IEC 61131-3: Programming Industrial Automation Systems*, 2nd edn, Springer-Verlag, Germany.
- Juran, J.M. 1999., *Juran's quality handbook* 5th edn, New York, McGraw-Hill Publishing Co.
- Kaner, C. 2006, "Exploratory Testing ", *Quality Assurance Institute Worldwide Annual Software Testing Conference* Florida Institute of Technology, USA.
- Kanstrén, T. 2010, *A Framework for Observation-Based Modelling in Model-Based Testing*, University of Oulu.
- Korel, B. 1990, "Automated Software Test Data Generation", *IEEE Transactions on Software Engineering*, vol. 16, no. 8.
- Kramer, U. 2001, "Continuous testing as a strategy of improving the PLC software development cycles", *IEEE/ASME International Conference on Advanced Intelligent Mechatronics* Como , Italy.
- Maher, P., Kourik, J. & Chookittikul, W. 2010, "Exploratory Study of Agile Methods in the Vietnamese Software Industry", *Fifth International Multi-conference on Computing in the Global Information Technology* IEEE, Valencia.
- Manolache, L. & Kourie, D. 2001, "Software testing using model programs", *Software-practice and experience*, vol. 10.
- McMinn, P. 2004, "Search-Based Software Test Data Generation: A Survey", *Software Testing, Verification & Reliability*, vol. 14, no. 2.
- Melnik, G. & Meszaros, G. 2009, *Acceptance Test Engineering Guide. Volume 1: Thinking about Acceptance*.
- Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A. & Teller, E. 1953, "Equation of state calculations by fast computing machines", *The Journal of chemical physics*, vol. 21, no. 6.
- Meyers, G.J. (ed) 2004, *The Art of Software Testing*, 2nd edn, John Wiley & Sons, New York.

- NI 2011, *Automating Real-Time Tests Using NI VeriStand and NI TestStand*. Available: <http://www.ni.com/> [citted on 22.08.2011].
- NI 2010, , *What is PXI?*. Available: <http://www.ni.com/> [cited on 25.08.2011].
- Pacheso, C., Ernst, M. & Eclat, D. 2005, *Automatic Generation and Classification of Test Inputs*, MIT Department of Electrical Engineering and Computer Science.
- Prather R., Myers J. P., Jr. 1987, "The path prefix software testing strategy", *IEEE Transactions on Software Engineering*, vol. 13, no. 7.
- Rothermel, G. & Harrold, M.J. 1996, "Analyzing regression test selection techniques", *IEEE Transactions on Software Engineering*, vol. 22, no. 8.
- Scacchi, W. 2001, "Process Models in Software Engineering" in *Encyclopedia of Software Engineering*, ed. J.J. Marciniak, 2nd edn, John Wiley and Sons, Inc, New York.
- Target, *Software Process Models*. Available: <http://www.the-software-experts.de/index.htm> [cited on 28.04.2011].
- Utting, M. & Legear, B. 2007, *Practical Model-Based Testing. A tools approach*, Morgan Kaufmann Publishers, Amsterdam.
- Winkler, D., Hametner, R. & Biffl, S. 2009, "Automation Component Aspects for Efficient Unit Testing", *14th IEEE international conference on Emerging technologies & factory automation* IEEE Press Piscataway, USA.
- Yaun, X. & Memon, A. 2007, "Using GUI Run-Time State as Feedback to Generate Test Cases", *29th international conference on Software Engineering* IEEE Computer Society, USA.

APPENDIX A. QML program of the crane power supply model

```
/** Main entry point to the model i.e. the place where the system "starts". */
void main()
{
    // Instantiate the 'Myproject' and start execution of the state machine.
    Myproject mc = new Myproject();
    mc.start("Myproject");

    /** Declaration of the 'Myproject' state machine.

        This state machine has its own execution thread and it supports
        communication with other state machines via ports. */
    class Myproject extends StateMachine
    {
        AnyRecord a = in.receive();

        protected void OUTput(boolean a, boolean b)
        {
            // Build user indication request with default values
            OutputSignal outputsignal;
            outputsignal.timerreset = a;
            outputsignal.craneon=b;
            out.send(outputsignal);
        }
    }

    /** Declaration of the external interface of the system being modeled. This is
        specific to system modeling; a similar construct does not appear usually in
        programming languages. In this "system block", we initially declare one inbound
        interface (in) and one outbound interface (out). The identifiers 'in' and
        'out' are the names for the interfaces in the model. After the colon we
        list the types of records that can possibly go through the interface in
        question. */
    system
    {
        Inbound in : MesCraneRequest,
        MesMainContactorEnergized,MesDriveRequest,MesCraneInterlocks;
        Outbound out : OutputSignal;
    }

    /** Declaration of a message type, which is technically presented as a "record
        type". It is a record of pure data. This record type 'MyMessage' is empty,
        i.e. it does not contain any actual data fields. */
    record MesCraneRequest {
        boolean b;
    }

    record MesMainContactorEnergized{
        boolean contactor;
    }

    record MesDriveRequest{
        boolean driverequest;
    }

    record MesCraneInterlocks{
        boolean interlocks;
    }

    record OutputSignal{
        boolean timerreset;
        boolean craneon;
    };
}
```

APPENDIX B. Java model of the crane power supply

```
/**
state 1 crane request 'CraneONReq'
state 2 main energized request 'MainEnergReq'
state 3 drive request 'DriveReq'
state 4 interlocks request 'InterLockReq'
state 5 timer set 'TimerSet'
state 6 crane on 'CraneOn'
state 7 timer reset 'TimerReset'
state 8 nothing 'Nothing'

action1 crane request off
action2 crane request on
action3 main contactor energized off
action4 main contactor energized on
action5 drive request off
action6 drive request on
action7 crane off interlocks on
action8 crane off interlocks off
action9 after timer set
*/
package nz.ac.waikato.modeljunit.examples;

import nz.ac.waikato.modeljunit.Action;
import nz.ac.waikato.modeljunit.FsmModel;
import nz.ac.waikato.modeljunit.RandomTester;
import nz.ac.waikato.modeljunit.Tester;
import nz.ac.waikato.modeljunit.coverage.CoverageMetric;
import nz.ac.waikato.modeljunit.coverage.TransitionCoverage;

/** Crane power supply example of a finite state machine (FSM) for testing.
*/
public class Crane implements FsmModel
{
    private String state = "CraneONReq";

    public Crane()
    {
        state = "CraneONReq";
    }

    public String getState()
    {
        return String.valueOf(state);
    }

    public void reset(boolean testing)
    {
        state = "CraneONReq";
    }

    public boolean action10Guard() { return state.equals("CraneONReq"); }
    public @Action void action10()
    {
        state = "Nothing";
    }

    public boolean action2Guard() { return state.equals("CraneONReq"); }
    public @Action void action2()
    {
        state = "MainEnergReq";
    }

    public boolean action3Guard() { return state.equals("MainEnergReq"); }
    public @Action void action3()
    {
        state = "TimerReset";
    }

    public boolean action4Guard() { return state.equals("MainEnergReq"); }
    public @Action void action4()

```

(continues)

APPENDIX B. (continues)

```
{
    state = "DriveReq";
}

public boolean action5Guard() { return state.equals("DriveReq"); }
public @Action void action5()
{
    state = "TimerReset";
}

public boolean action6Guard() { return state.equals("DriveReq"); }
public @Action void action6()
{
    state = "InterLockReq";
}

public boolean action7Guard() { return state.equals("InterLockReq"); }
public @Action void action7()
{
    state = "TimerReset";
}

public boolean action8Guard() { return state.equals("InterLockReq"); }
public @Action void action8()
{
    state = "TimerSet";
}

public boolean action9Guard() { return state.equals("TimerSet"); }
public @Action void action9()
{
    state = "CraneOn";
}

/** This main method illustrates how ModelJUnit can be used
 *  * to generate a small test suite.
 */
public static void main(String args[])
{
    // create our model and a test generation algorithm
    Tester tester = new RandomTester(new FSM());

    // build the complete FSM graph for our model, just to ensure
    // that we get accurate model coverage metrics.
    tester.buildGraph();

    // set up our favourite coverage metric
    CoverageMetric trCoverage = new TransitionCoverage();
    tester.addListener(trCoverage);

    // ask to print the generated tests
    tester.addListener("verbose");

    // generate a small test suite of 50 steps
    tester.generate(50);

    tester.getModel().printMessage(trCoverage.getName() + " was "
    + trCoverage.toString());
}

}
```