Master Thesis

Examination of the Automated Test System for AVL IndiCom

Hannes Schneider, Bakk. rer. soc. oec.

Institute of Software Technology (IST)
Graz University of Technology
Head: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Wolfgang Slany

Assessor: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa

Graz, April 2015

Abstract

The development of new software or new features for software applications requires an accurate inspection of the compliance to the quality standards. Modifications of the software can lead to new bugs and malicious side effects as well as to failing the desired requirements. To work against such regressions, automated test systems are used during software development. Such systems often have a high complexity and have to be customized to the needs of the application. Regression testing plays a major role for the maintenance and is a critical success factor for most software projects. The software development process of the combustion analysis program "AVL IndiCom" applies such an automated testing solution. However, practice has shown that the constant evaluation and maintenance of the results leads to a high effort. This diploma thesis reveals the weakness of this system and shows how improvements to the automation can be implemented.

Kurzfassung

Die Entwicklung von neuer Software oder neuen Features für den Anwendungsbereich erfordert eine genaue Überprüfung der Einhaltung von Qualitätsanforderungen. Modifikationen an der Software bergen stets die Gefahr von Fehlern, unerwünschten Nebeneffekten sowie die Nichteinhaltung der verlangten Anforderungen. Um solchen Rückschritten entgegen zu wirken werden verstärkt automatisierte Testverfahren in der Softwareentwicklung eingesetzt. Solche Systeme sind jedoch oft sehr komplex und müssen meist auf die zu testende Anwendung maßgeschneidert werden. Der Regressionstest spielt speziell in der Wartung eine große Rolle und die wirtschaftliche Durchführung des Regressionstests ist in vielen Projekten ein kritischer Erfolgsfaktor. In der Entwicklung des AVL Verbrennungsanalyse Programms "AVL IndiCom" wird eine solche automatisierte Testlösung eingesetzt. Allerdings hat sich gezeigt, dass die ständige Wartung und Auswertung der Ergebnisse mit sehr hohem Aufwand verbunden sind. Diese Arbeit zeigt die Schwächen dieses Systems auf und stellt einige Verbesserungen der Testautomatisierung vor.

Acknowledgments

First I want to thank my family, especially my mother, for their support throughout my whole life and the opportunity to study. Without them this would not be possible.

I would like to thank all the people from MIS at AVL List GmbH in Graz who supported my work. In particular, I would like to thank Mr. Anton Semlitsch for the great support of my work and the supervision of this diploma thesis.

Graz, April 2015 Hannes Schneider

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

AVL is a company dealing with the development of powertrain systems, with internal combustion engines as well as instrumentation and test systems. Processes in the combustion engine are primarily not calculated with complete thermodynamic models, but are measured systematically. By varying certain parameters like values of the engine electronics, the combustion engine can be optimized. A variation of a wrong parameters can lead to a serious harm of the engine, therefore the correctness of the measured data is essential for the analysis of the combustion engine. A precise examination of the measurement data has to be done, before the measurement software is delivered to the customer. The evaluation of the measured data is a challenging and time consuming task, therefore an automation of this process has been implemented.

1.1 Motivation

My first contact with test automation was during my employment for AVL as a trainee and then as a part-time job during my studies. My task covered in particular the maintenance of the automatic test system, the analysis of detected errors and the reporting of these errors. The introduction of new agile software development methodologies and the increasing maintenance effort for the automatic test required an examination of the current automated test methods.

1.2 History

The term of "test automation" is a comprehensive term. Many test requirements can be implemented in many different ways. Test automation can be applied in different test phases, from component test to acceptance test. Test automation can be applied to different activities, from automatic test case generation to automatic execution of test plans. Therefore no uniform recipe for implementing an automated test system can be found.

With the reasoned increase of new software development methodologies and shorter release cycles, automated testing is becoming more and more important. Test automation can also help to meet the "PI Objectives"[1] and therefore contributes to the business value.

1.3 About this document

This master thesis summarizes the results of my work. Subsequently, a coarse overview of the topics in this thesis is presented.

The term combustion analysis is not a commonly known term in the field of computer science and originates from the field of mechanical engineering. Therefore, in chapter 2, the fundamentals about combustion analysis will be explained and the functionality of AVL IndiCom which is relevant for the test automation will be described. In chapter 3 the characteristics of the current implementation of the automatic test system will be explained. Chapter 4 will focus on the current automatic test system and analyse how the implementation meets the requirements. The perceptions found in chapter 4 required a test of the robustness of the current test evaluation method, which will be shown in chapter 5. The main work of this master thesis was to find improvements for the test automation. This expertise will be illustrated in chapter 6. At the end of this thesis I will provide the results of the field test in chapter 7.

[1] Program Increment Objectives, specific business and technical goals

Chapter 2

Combustion Analysis with AVL IndiCom

The combustion analysis of an engine depends basically on the evaluation of the anticipated pressure chart and expected work based on the gas laws. Before Nikolaus August Otto [1] launched the first four-stroke engine recorded on 18 May 1876, he calculated the pressure chart and with this was able to confirm his calculations by measuring the cylinder pressure on his test engine. These mechanically recorded pressure traces are called indicator charts.[see Pis02, chap. 1]

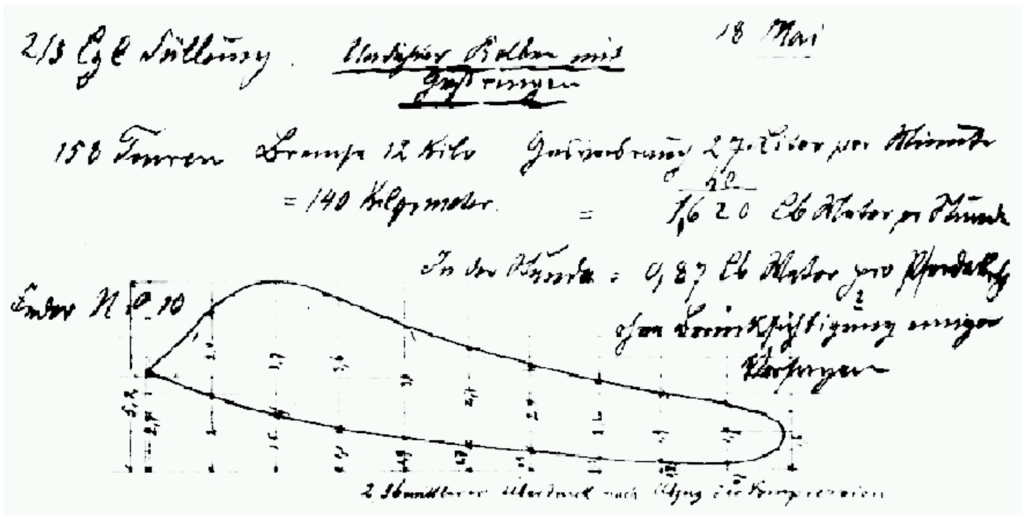


Figure 2.1: *Indicator chart recorded by Nikolaus August Otto (from Friedrich Sass: "Geschichte des deutschen Verbrennungsmotorenbaus von 1860-1918")*

The engine instrumentation since then has changed considerably, however the concept of "engine indicating" still remains. The crank angle-based measurement of

[1] Nikolaus August Otto (10 June 1832, Holzhausen an der Haide, Nassau - 26 January 1891, Cologne) was the German engineer of the first internal-combustion engine

the pressure inside the cylinder in relation to the instantaneous position of the piston is still a central parameter to understand the inside engine phenomena. Nowadays the field of applications for combustion measurement has a wide range. Piezoelectric pressure transducers and computer-supported data acquisition allows us to obtain extensive information from the analysis of measured pressure curves.

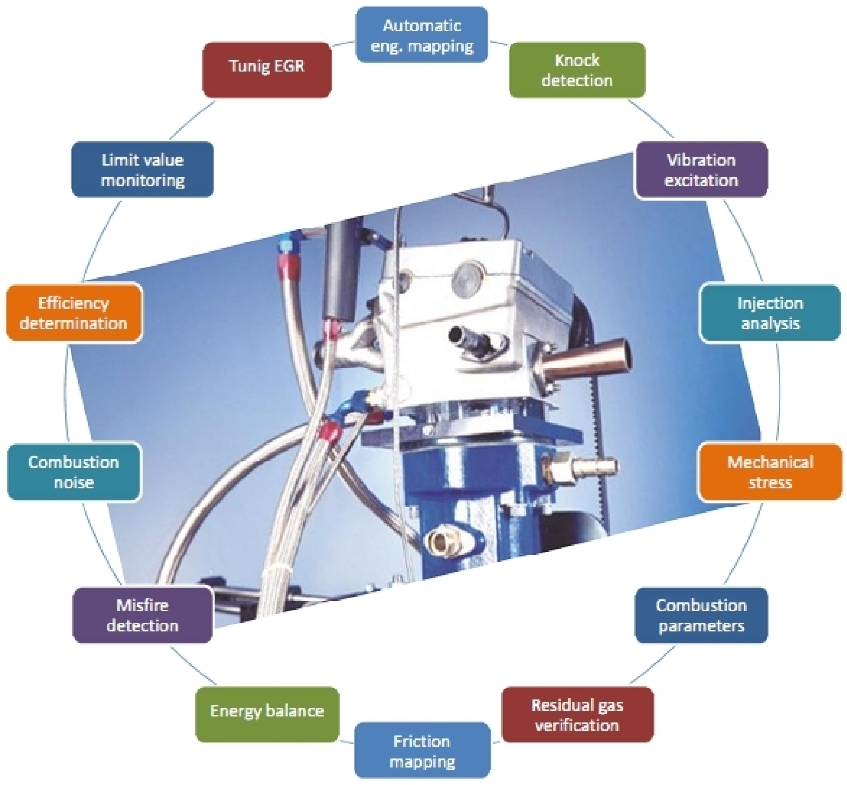


Figure 2.2: *Application areas of indicating technology*

AVL IndiCom is an application designed for data acquisition, evaluation and result transfer of crank angle based and time based measurement values from combustion engines.

2.1 Data acquisition

This chapter will give a brief overview of the functioning of AVL IndiCom. The first step to retrieve a measurement is to supply the engine with sensors and connect it to a high-performance electric generator. With this generator the workload can be varied as required. The sensor data is transmitted to the indication hardware and then to a PC on which the data is evaluated by AVL IndiCom.

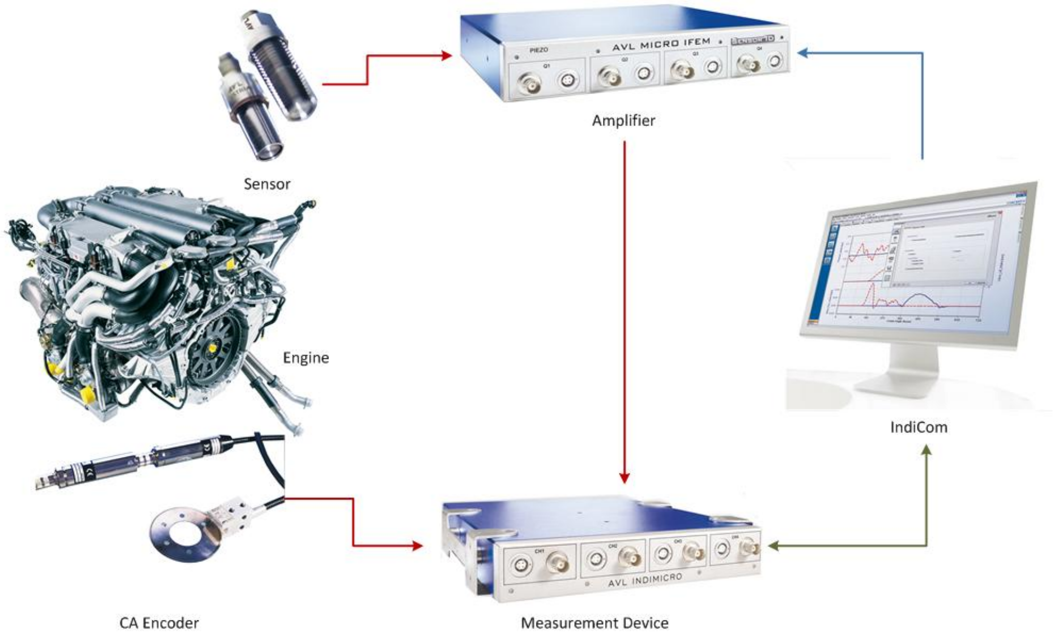


Figure 2.3: *Indicating measurement chain*

AVL IndiCom provides two different measurement modes which will be relevant for the automatic test. One is the single measurement over a defined number of cycles and the other is the continuous acquisition of data.

2.1.1 Single Measurement

A single measurement (procedural measurement) is terminated automatically after the defined measurement duration. The recorded cycles can then be viewed one by one. [see Gmb11c, chap. 11]

2.1.2 Continuous Acquisition

In a continuous acquisition, curves and results are displayed continuously. The data is continuously recorded in a ring buffer, and the most recent cycle is displayed. This produces an oscilloscope-type display. This measurement mode is stopped manually or can be switched over to a single measurement. The last cycles remain available in the memory. [see Gmb11c, chap. 11]

2.2 AVL IndiCom Components

AVL IndiCom is a state-of-the-art user interface and control software for all AVL indicating systems. Furthermore, this software also provides the possibility to acquire data from other systems such as NI-DAQ card, CAN buses of ASAP3. Depending on the used devices and options the components in the following chapters will be available. Figure 2.4 shows the main components of AVL IndiCom and in the following subsections the relevant components for the automatic test will be described. Basically, all calculations available with AVL IndiCom depend on the measured signals based on parametrization of the measurement.

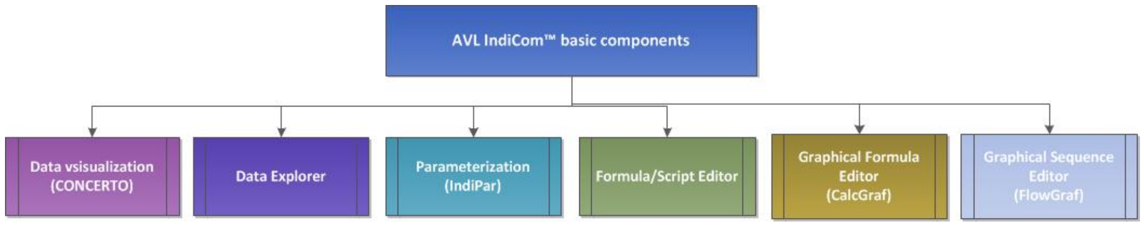

AVL IndiCom™ basic components
Data vsisualization (CONCERTO)
Data Explorer
Parameterization (IndiPar)
Formula/Script Editor
Graphical Formula Editor (CalcGraf)
Graphical Sequence Editor (FlowGraf)

Figure 2.4: *IndiCom Components*

2.2.1 Parametrization

Before a measurement can be started, the utilized hardware, such as an indicating or an simulated device, has to be parametrized. The software automatically detects the connected devices and amplifiers. A graphical view of the devices shows the available connections on the connected devices. This includes e.g. settings for crank angle marks, engine geometry, TDC values, signal names and types, measurement ranges and resolutions and calibration values. These settings can then be saved to a file and can be loaded later [see Gmb11c, chap. 6]. The parametrization is necessary for each measurement and therefore an essential part of the automatic test.

2.2.2 Formula/Script Editor

The Formula/Script Editor provides a complete development environment within AVL IndiCom. Predefined control sequences are stored in Script files and may contain a number of commands to be interpreted and executed by AVÃ– IndiCom. In this way, in addition to the standard functions, AVL IndiCom provides a full-fledged programming environment that can be used for developing individual applications. [see Gmb11a, chap. 1] The script language is very similar to the interpreter language VB and the Script Editor also contains debugging tools for the development. This component is essential for the automatic test, as most of the IndiCom functionality can be operated with these scripts.

2.2.3 Graphical Formula Editor

CalcGraf is an intelligent formula interpreter which can be parametrized via a graphical editor. CalcGraf can be used to generate graphic calculation models where various function blocks are linked and wired to inputs and outputs. Formula files are generated for all outputs while the model is subsequently compiled. These formulas act as virtual channels and can be selected in the channel list of an indicating data file for use in diagrams and tables. When the formula is compiled, the outputs of the model will be added to the output channel list and the calculation of those channels will be executed when the channel is displayed. [see Gmb11b, chap. 5.7]

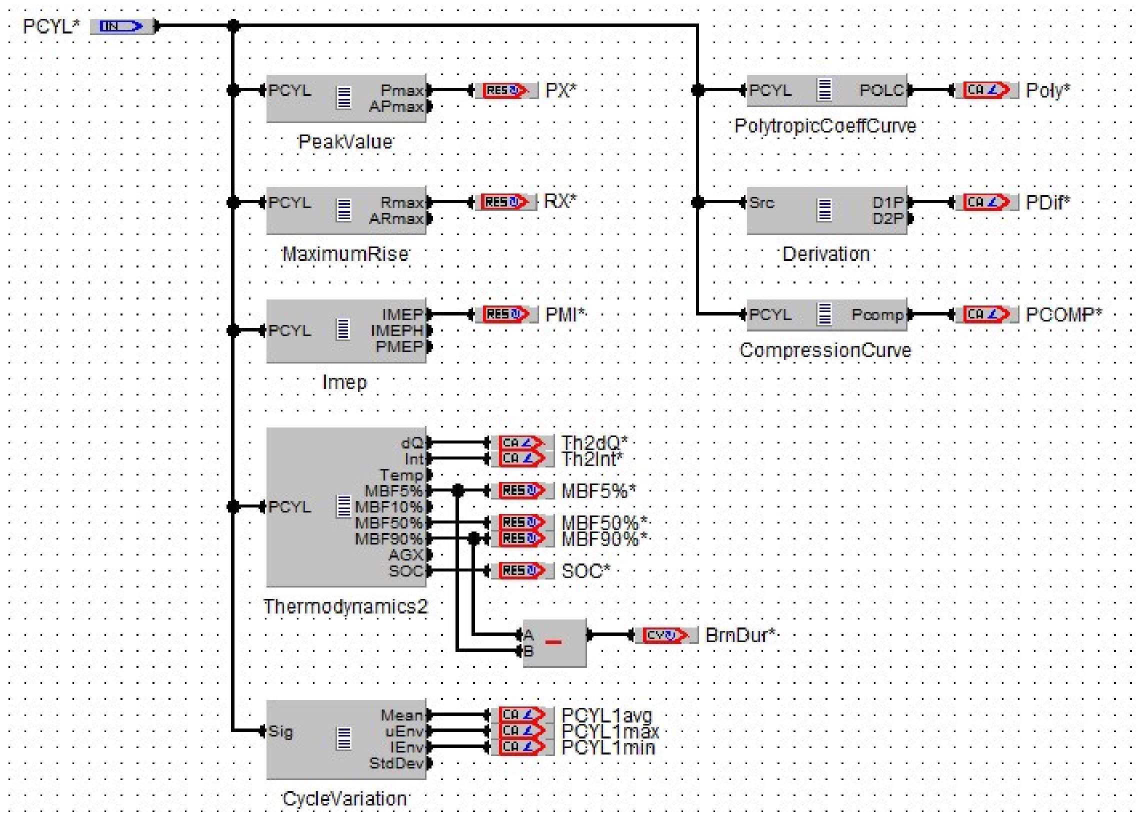


Figure 2.5: *CalcGraf Formula - Model for calculation of the cylinder pressure*

The output of each model will be a script which simulates a channel in the measurement. These calculated results will also be checked by the automatic test.

Chapter 3

Characteristics of the automatic test system

As described in the previous chapter, one of the basic functionalities of IndiCom is the data acquisition and processing from various measurement devices. These devices are normally plugged to different sensors on a combustion engine. Changes on the data acquisition code can influence all results calculated by the software.

Since the test set-up for a combustion engine is not available during development, the first consideration was to build a device, which can simulate the signals produced by a combustion engine. A hardware element was developed which can playback signals captured from a real combustion engine was developed. In the beginning of the development of this software, all signals were checked simultaneously by a manual measurement with an oscilloscope and then compared to the measured data on the PC. This mostly is an extensive task and also requires a detailed knowledge on combustion analysis.

To reduce the effort of manual testing, the next step was to automate this task and build a regression test system. A regression test is not a common test method, in fact it is the repetition of already completed tests. After a change to the program code it is possible to detect side effects by repeating the tests and comparing the results.

The principal of the regression testing is trivial. Two different versions of the software need to be tested, an older version which is already tested and approved and the current version. On behalf of the specification, the test results of the reliable version are compared to the results of the current version. If the results of the current version match the results of the reference version, this implies that the behaviour of the software is the same and no regression was found. [see Pre07, chap. 8]

3.1 The test set-up

To execute an automatic test, the test environment has to be set up. This environment consists of a PC running the latest version of AVL IndiCom. The PC is connected to the Engine Simulator ant the simulator is connected to the measurement device. Optionally, the measurement device can be connected to an amplifier. Depending on the measurement hardware, the device is connected to the PC via RJ45, USB or Firewire (see figure 3.1).

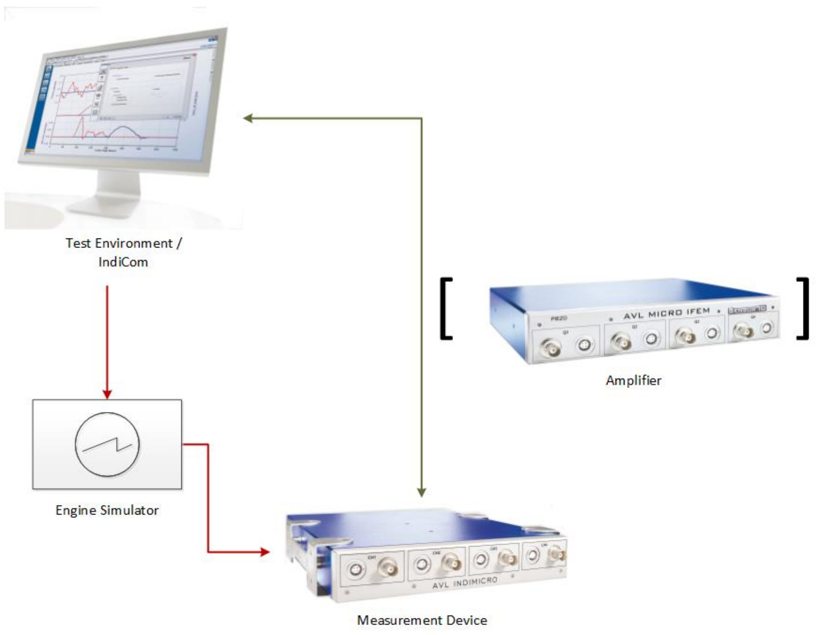


Figure 3.1: *The test enrivonment*

Over the years, new measurement devices were developed and the support for new devices was implemented into AVL IndiCom. Different specifications of the functionality and measurement accuracy of the measurement devices, made it necessary to build multiple test set-ups.

3.2 Test preparation

To execute the automatic test routine, some manual preparation steps have to be fulfilled first:

1. Check for the latest version of AVL IndiCom
2. Copy the setup files to a local directory
3. Copy the license files to a local directory

4. Set configuration entries in the setup INI file
5. Execute the AVL IndiCom setup
6. Checkout the test environment to the installation folder
7. Set configuration entries in the application INI file
8. Start AVL IndiCom
9. Exchange the standard work environment with the test environment
10. Start the test routine

These steps have to be repeated for every new AVL IndiCom build and for each measurement device on different PCs.

3.3 The test sequence

To accomplish the automatic test, different steps have to be executed. This procedure consists of a manual task, the test preparation, and an automatic task, the test execution.

3.3.1 Test execution

Each test is implemented as an AVL IndiCom script which basically consists of the same set of instructions.

Figure 3.2 shows the control sequence for one automatic test case. As a first step the parametrization for the measurement has to be loaded. This file defines which channels will be measured and determines the resolution of the measured data. Then a first communication with the measurement device is initiated.

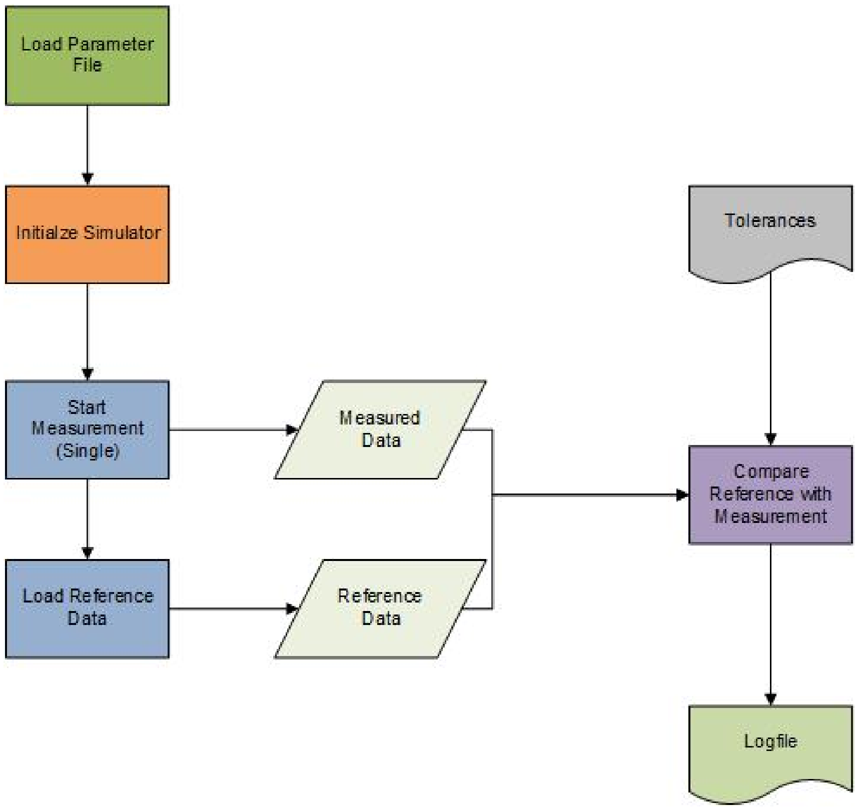


Figure 3.2: *Control sequence for a single test*

In the next step the Engine Simulator is initialized. A file containing the recorded data from a measurement on a real combustion engine, is transmitted to the simulator and then the simulation is started.

Now AVL IndiCom starts to communicate with the measurement device and the measured data is transferred to the PC. Depending on the measurement mode, the measurement will stop after a defined time period or a certain set of cycles.

3.3.2 Test evaluation

After the measurement has stopped, the reference data set and the tolerance values are loaded. Due to timing differences between the simulator and environmental influences on the measurement devices, an exact comparison of the measured signal with the reference signal is not practicable.

Single measurement

A single measurement always contains the same amount of data, e.g. one value per crank angle. For this use case a statistical evaluation of the measured signal is made.

$$\overline{x} = \overline{x}_n := \frac{1}{n}\sum_{i=1}^{n} x_i \tag{3.1}$$

$$\sigma_x = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{(n-1)}} \tag{3.2}$$

$$x_{min} = \min(x_1, x_2, ..., x_n) \tag{3.3}$$

$$x_{max} = \max(x_1, x_2, ..., x_n) \tag{3.4}$$

Equation (3.1) is the arithmetic mean of the measured data. Equation (3.2) is the standard deviation of the data set and the equations (3.3) and (3.4) are the highest and lowest measured values. [HEK05]

Figure 3.3 shows the comparison of a measured data set and the loaded reference data set. Now the statistical evaluation of both signals is compared. As a standard value a deviation of +/- 1 % of the reference value is defined in the tolerances file. This condition has to be satisfied by all values, so that a test is considered as passed. The standard deviation represents an exception. Because values can possibly become very small, an absolute deviation of 0.05 is defined. This means, that if the calculated standard deviation of the reference signal is lower than this value, the deviation will be +/- 0.05.

	Reference	-1%	$+1\%$	Measured	Pass
x_{min}	44.800000	44.352000	45.248000	45.100000	TRUE
x_{max}	3515.200000	3480.048000	3550.352000	3517.900000	TRUE
$\overline{x}$	382.192500	378.370575	386.014425	384.860139	TRUE
σ_x	655.720667	649.163460	662.277873	655.782997	TRUE

Table 3.1: Test evaluation for cylinder pressure [percent], successful test Evaluation of the graph shown in in 3.3:

Table 3.1 shows the evaluation for the signals presented in figure 3.3. Since this comparison method is highly sensitive to any environmental influences, like minimal variations in the environment temperature, a definition of absolute tolerances was necessary. Another problem is that the percentage depends on the value of the reference, and if this value is very low, the tolerance is even lower.

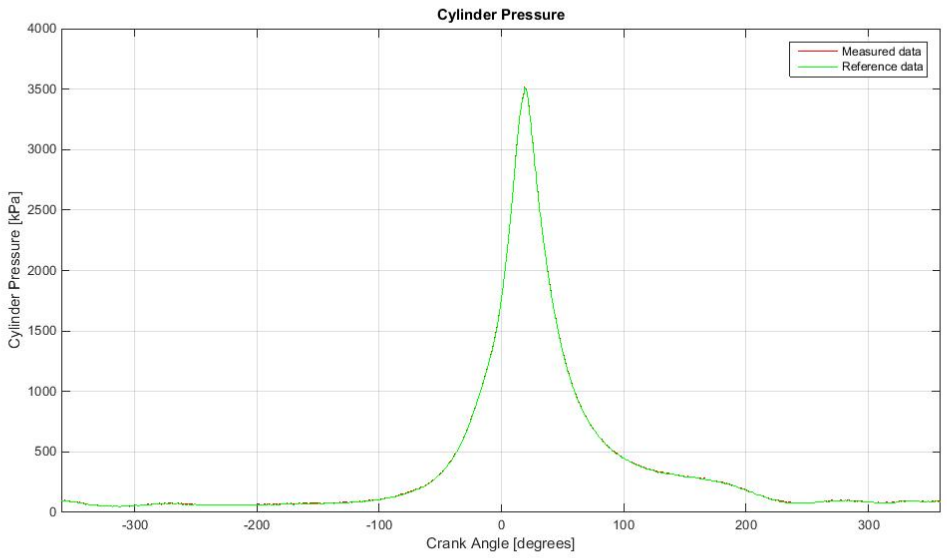


Figure 3.3: *Measured signal and reference signal of cylinder pressure*

Temperature fluctuation can influence the operating temperature of the measurement devices and the integration of new measurement devices into the automatic test can lead to substantial variations of the measured data. Due to this fact, the introduction of absolute tolerance values was inevitable.

Figure 3.4 shows the result of a failed test, which was created with the AVL IndiModul 621 device and measured with the new IndiCom Mobile device.

	Reference	-1%	$+1\%$	Measured	Pass
x_{min}	44.800000	44.352000	45.248000	43.645444	FALSE
x_{max}	3515.200000	3480.048000	3550.352000	3515.477658	TRUE
$\overline{x}$	382.192500	378.370575	386.014425	382.203308	TRUE
σ_x	655.720667	649.163460	662.277873	655.698394	TRUE

Table 3.2: Test evaluation for cylinder pressure [percent], failed test
Evaluation of the graph shown in figure 3.4

This test will fail since the x$_{min}$ value is not in the tolerance bound of +/- 1 % of the reference minimum. For this reason, the tolerances can also be expressed as absolute values. These values can be defined, for each test and measurement channel in the tolerances file.

The tolerances are defined in the so called initialization file, which basically is a text simple file. Adapting these tolerances requires high knowledge of the measured signal, and the administration of these values is causing more and more effort.

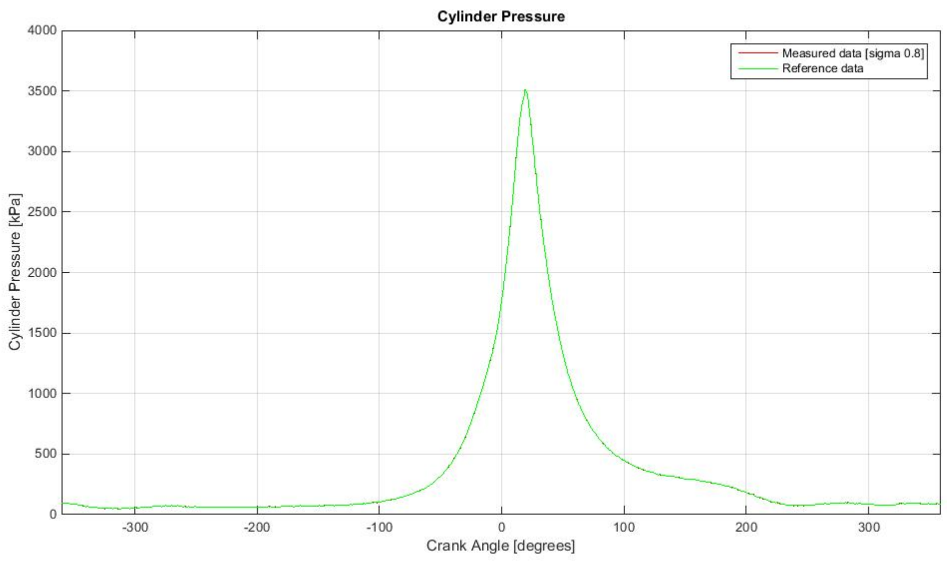


Figure 3.4: *Measured signal and reference signal of cylinder pressure*

	Reference	Tolerance	-Tolerance	+Tolerance	Measured	Pass
x_{min}	44.800000	1.17	43.63000	45.970000	43.645444	TRUE
x_{max}	3515.200000	0.28	3515.47766	3550.352000	3515.480000	TRUE
$\overline{x}$	382.192500	0.02	382.20330	386.014425	382.212500	TRUE
σ_x	655.720667	0.03	655.69839	662.277873	655.750667	TRUE

Table 3.3: Test evaluation for cylinder pressure [absolute]
Test evaluation with absolute tolerance values for figure 3.4

Continuous measurement

During a continuous measurement the measurement signals are recorded by the hardware and processed by a real time processor. This RTP data is continuously transmitted to the PC. Depending on the computing power, signal count, resolution, revolution speed and the defined calculations, the transmission of the data sometimes is not possible. In this case, only the raw data of the current operating cycle is transferred to the PC and continuously refreshed. All other measurement results will be calculated when the measurement has finished and all data has been transferred.

Since the Engine Simulator and the PC are not synchronized, the start and stop of the measurement can vary which influences the measurement result. Furthermore, the values calculated after completion of the measurement, depend on the raw data transferred and this impedes the comparison of those signals. The statistical evaluation introduced in this chapter will fail in most cases for the described measurement type. For this reason a sinus simulation record was created and this data will be used for the time-based measurements.

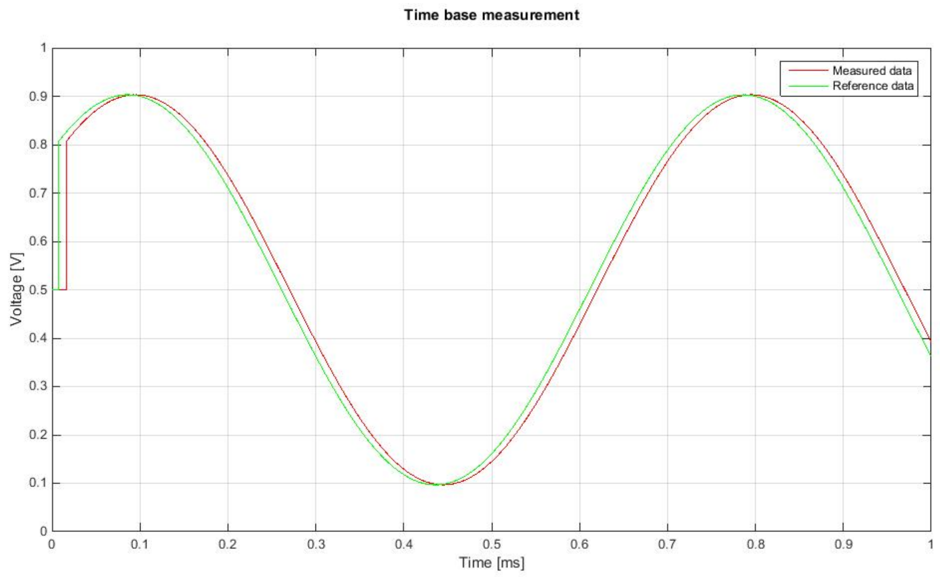


Figure 3.5: *Measured time based signal and reference signal (resolution 0.025 sec)*

Figure 3.5 shows the comparison of a measured time-based result data set and its reference data set, with a time delay of 2.995 seconds. The start of the continuous measurement depends on the current workload of the PC and the communication with the measurement device. This delay is not predictable and normally ranges between 0 and approximately 3 seconds.

For this reason, time-based signals will be compared by an absolute deviation, which is a separate value in the tolerances file. To compare two time-based signals, the first common values of the two signals have to be found by searching the nearest rising or falling edge. Then the translation between the signals is calculated and added to the measured signal. Now, each single value of the measured data set is compared to its corresponding value in the reference signal.

$$x_{ref(Min)}(i) = x_{ref}(i) - \mu \tag{3.5}$$

$$x_{ref(Max)}(i) = x_{ref}(i) + \mu \tag{3.6}$$

$$x(i)_{Pass} = (x_{comp}(i+\delta) > x_{ref(Min)}) \text{ AND } (x_{comp}(i+\delta) < x_{ref(Max)}) \tag{3.7}$$

Equation (3.7) shows the evaluation of a time based signal. In the equations (3.5) and (3.6) the tolerance range is calculated, where μ is the tolerance value loaded from the tolerances file. The value δ is the translational displacement of the two signals. If the condition in (3.7) is not fulfilled, this test will fail.

3.4 Endless test

Each single test is implemented as a script file which is executed with AVL IndiCom. These test files are located in a folder in the "AutoTest" environment in the AVL IndiCom program folder. In the endless test mode all scripts from the "AutoTest" folder are executed one after another in alphabetical order of the script name.

3.5 Reporting

The result of each single automatic test is recorded to a text file. This Log file will also contain the execution time and duration of each test, as well as a primitive evaluation of the memory consumed by this test. After the execution of all tests a summary showing the passed and failed tests is displayed.

There is no automatic reporting to the test department or to the development department. The results are checked after each run and failed tests will be executed again manually. If the failed test is not reproducible, it will not be considered as error. In case of a reproducible error, the test result will be entered into JIRA and a bug in the software is reported.

Chapter 4

Analysis of the automatic test system

The securing of the correctness and reliability of software is vitally important to the product life cycle of a software product. The cost trend shows a marked increase of the software costs in comparison to the hardware costs. Hence, a cost reduction in the area of software development is highly economic.

The analysis of the costs in the software development process leads to the result, that the biggest portion of the expenses arises during the maintenance phase of a software product which is already on the market. This is the consequence of inadequate quality of the software. The cause are errors which originate during the software development and which are found when the product is used by a customer. If the source code is also poorly structured and insufficiently documented, the error detection and correction can be a very expensive task. [see Lig09, chap. 1]

Software quality can be defined by many different characteristics. Common definitions are 'conformance to the requirements' or 'fitness for use', but these definitions do not provide a mechanism to judge better quality when two products are equally fit to be used [O'R14].The International Organization for Standardization defines six quality characteristics which can be seen in table 4.1.

Characteristic	Description
Functionality	This indicates the extent to which the required functionality is available in the software
Reliability	This indicates the extent to which the software is reliable.
Usability	This indicates the extent to which the users of the software judge it to be easy to use.
Efficiency	This characteristic indicates the efficiency of the software
Maintainability	This indicates the extent to which the software product is easy to modify and maintain.
Portability	This indicates the ease of transferring the software to a different environment

Table 4.1: ISO-9126 Quality characteristics
[see O'R14, Table 1.1]

Since the automatic test itself is a software product, there should be the same demands on quality as for a commercial software product. In this chapter I would like to investigate, how the automatic test meets the quality characteristics described in ISO-9126.

To evaluate the extent to which the above mentioned characteristics are met in the automatic test, first the requirements have to be defined. Since no specification and no documentation is available, the requirements defined in retrospect as follows:

1. The test should find errors in data acquisition
2. The test should find version conflicts
3. The test should run with different measurement devices
4. Different measurement types should be evaluated
5. The test should run as endurance test
6. The test should provide data for performance analysis
7. The system should be easy to maintain
8. Everybody should be able to create new test cases
9. It should be possible to reproduce errors
10. Different test plans should be executable
11. Everybody should be able to understand the test cases
12. The test results should be evaluated and easy to handle

4.1 Functionality

The points 1, 2 and 4 describe the functionality of the automatic test. In the following these points will be focused and examined, int terms of how the automatic test meet those requirements.

4.1.1 Finding errors in the data acquisition

As described in section 3 each single test is executing a measurement and then compares the results to a reference data set. A deviation from the reference will result in an error and as consequence the test has failed. Related to this functional requirement, the false positive and false negative detection rate has to be examined.

False Negatives

A false negative error occurs when a test fails when it actually should not fail. Unfortunately this happens very often. New measurement devices, loose wiring and especially environmental influences on the hardware influence the measurement results. These circumstances produce slight measurement errors which lead to the fact, that many tests will fail. Especially the x_{min} and x_{max} are sensitive to these influences. Increasing the tolerances is only a temporary solution and will not solve the problem.

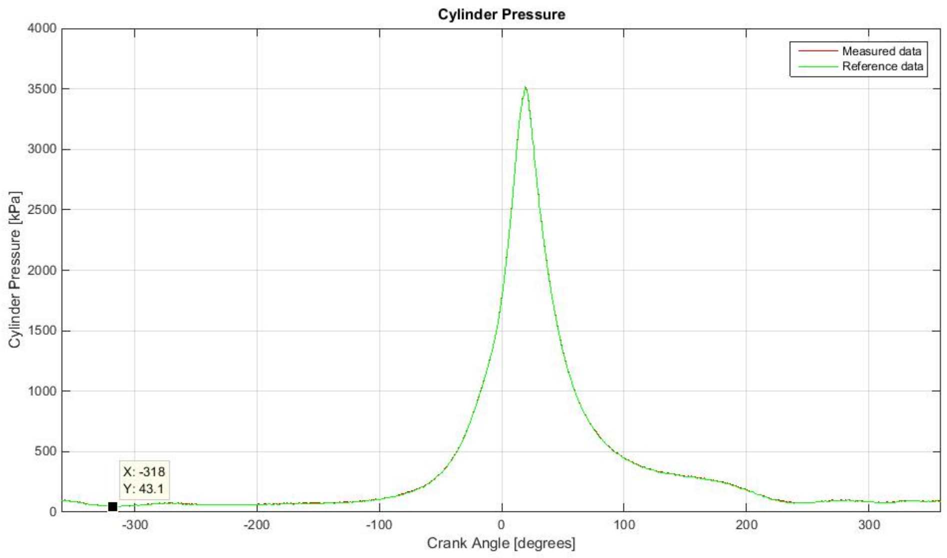


Figure 4.1: *Failed test due to one outlier*

Figure 4.1 and table 4.2 show the result of a failed test due to one minimal outlier. This result is not reproducible and a repetition of this test will most likely produce correct results.

	Reference	-1%	$+1\%$	Measured	Pass
x_{min}	44.800000	44.352000	45.248000	43.100000	FALSE
x_{max}	3515.200000	3480.048000	3550.352000	3517.900000	TRUE
$\overline{x}$	382.192500	378.370575	386.014425	384.857361	TRUE
σ_x	655.720667	649.163460	662.277873	655.784443	TRUE

Table 4.2: Test evaluation for cylinder pressure [percent], failed test
Evaluation of the graph shown in in 4.1:

This behaviour shows that the evaluation method is not robust to false negative errors and increases the immense effort during evaluation of the test results. A detailed determination of the robustness of the current evaluation method will be shown in chapter 5.

False Positives

A false positive error is a result, that indicates the given test has passed, when it actually should not have passed. This case definitely does not comply to the requirement. For some signal types this measurement evaluation is inadvisable.

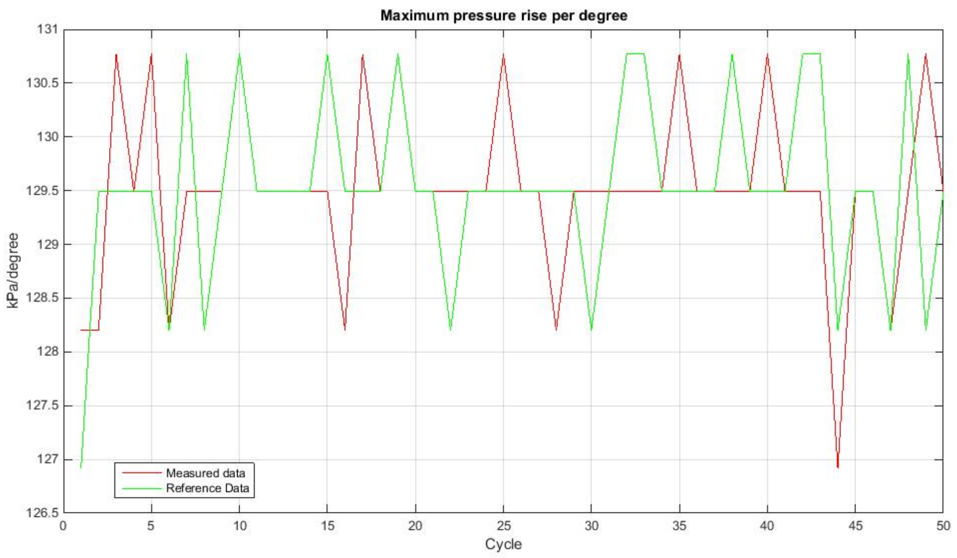


Figure 4.2: *Passed test with not correlated signals*

Figure 4.2 shows the maximum pressure rise per degree for a set of 50 cycles. Table 4.3 shows that for the measured data, which is considerably different to the reference signal, the statistical values are in the tolerance scope. The reason is, that the values allow no inference to the signal progression of the two signals. This shows that the statistical evaluation is appropriate for very similar expected signals, like in the case of cycle-based signals.

	Reference	-1%	$+1\%$	Measured	Pass
x_{min}	126.920000	125.650800	128.189200	126.920000	TRUE
x_{max}	130.770000	129.462300	132.077700	130.770000	TRUE
$\overline{x}$	129.514000	128.218860	130.809140	129.514200	TRUE
σ_x	0.840418	0.790418	0.890418	0.799309	TRUE

Table 4.3: Test evaluation for maximum pressure rise per degree [percent]
Evaluation of the graph shown in in 4.2:

4.1.2 Finding version conflicts

As already mentioned in chapter 3 the automatic test was developed for regression testing. The reference files provided in the test environment were created with an older and considered stable version of AVL IndiCom. As shown in the previous chapters a deviation from these references leads to an error and a test failure.

The development of new features and calculation methods in the software and the introduction of new measurement devices, leads to a lack of reference files. Since the creation of such references is an expensive task and the references have to be reviewed by an expert, this task has been neglected in the passed years.

The evaluation of the test results of the automatic test system showed, that no faultless test run was completed in the last month. The reason was not that the software had so many bugs, but the test environment was not stable enough.

At an average release cycle of the software of 1 to 2 versions per week no version conflicts could be found.

4.1.3 Evaluation of different measurement types

The automatic test should be able to evaluate as many different signal types as necessary. Currently, the main focus of the test are cycle based signals. As shown in chapter 3.3.2 the evaluation of time-based signals at the moment is limited.

as section 4.1.1 demonstrated, poorly correlated signals fail the tolerance check. This is essentially the case for very short signal periods with a very small value spread. Therefore, the evaluation of different measurement types too, does not comply with requirement.

4.1.4 Performance analysis

During the execution of all tests a log entry in the test result file is created with the currently consumed private memory of the AVL IndiCom and AVL IndiPar process. New data transfer methods implemented in AVL IndiCom lead to a heavy memory load. Transferring signals in a high resolution in real time processing mode can lead to out of memory exceptions and as a consequence, to a program crash. Bugs in the data transfer lead to inaccurate measurement results and performance impacts.

All tests scripts belonging to the automatic test are executed in the same order for each test run. This impedes the discovery of memory problems because most problems occur when tests with a high measurement resolution and therefore high data amounts are executed one after another.

The current evaluation of the recorded performance data showed, that these values are insufficient for a memory analysis. Furthermore, the runtime of the single tests is not recorded in an evaluable format. For this reason variations in the runtime of single tests can not be detected.

4.2 Reliability

The term software reliability is defined as the probability that the program will work according to its requirements and for a given amount of time [see Pha00, chap. 1]. In this section, the requirements mentioned in point 5 will examined. The continuous monitoring of the automatic test and the runtime are key features of any automated test environment.

4.2.1 Endurance test

The so-called endless test mode of the automatic test system executes all tests in the order of the test name and after all tests have been executed, the same order is repeated again, until the system is stopped manually or the program crashes.

Since the test script is implemented and executed in the software to be tested itself, a crash of the software leads to the abortion of the endless test. The version that is tested in most cases is a development version. Hence, errors and crashes are very likely. Outside office hours no complete run of all tests is available next day.

4.2.2 Test runtime

The period of time between the release build of the software and supply of the software to the customer may not exceed three days. Currently the automatic test has an average runtime[1] of 48 hours.

Since each test depends on the hardware and measurement time or defined cycles, there is not much room for improvements. The evaluation of the measurement results is computed in O(n) steps and is insignificant for the duration of a single test.

[1] Depending on the amount of test cases

4.3 Usability

The ISO 9126 standard defines usability as “a set of attributes that bear on the effort needed for use and on the individual assessment of such use, by a stated or implied set of users.” It then proposed a product-oriented usability approach. Usability was seen as an independent factor of software quality and it focused on software attributes, such as the interface, which makes it easy to use [see AKSS03, chap. 3.2].

The point 8, 9 and 11 are focusing on the usability of the automatic test and will be examined in the following sections.

4.3.1 Creation of new tests

The automatic test depends mainly on the simulation data provided by customers. These simulation files are very hard to get, since the combustion analysis is primarily used for engine prototypes. All available simulation files are already included in the automatic test and the creation of new tests is rarely necessary.

A document that describes the steps to create a new test case exists, however this is only usable for people with programming skills. Since most people concerned with the automatic test are specialists in indication technology, this can be a challenging task. Furthermore, in this case, the requirement can be considered as not fulfilled.

4.3.2 Reproduction of errors

An automatic test is only valuable when its results can be reproduced. As mentioned in chapter 4.1.1 the automatic test results in too many “false negative” findings. Most of these findings were produced through temporary environmental influences and are not reproducible.

4.3.3 Test plans

During the development of the automatic test, only one measurement device was produced by AVL. The growth of the market and the great demand by the customers lead to a diversification in the engine indication measurement sector. New measurement devices were developed and the software was adopted to comply to these devices.

As described in chapter 3.4 the execution order of the tests is based on the script file name. Since there is only one central folder from where the tests are executed, the changing of the order is very time-consuming. Therefore, the modification of the execution order or even building test plans is not possible.

4.3.4 Understandability of test cases

For analysing the test results, it is indispensable that the test cases are well documented. For this reason, a document containing all test cases with their name, simulation file, measurement resolution and expected results was created.

The experience showed that the information contained in this file is not sufficient to retrace the meaning of the test case. In addition, each test case contains the same information in its script file. Hence, this information is redundant and has to be maintained twice. To understand the meaning of a test case, additional information like a description or the measured channels and the amount of cycles is absolutely necessary.

4.4 Efficiency

Efficiency is "the capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions". Efficiency is divided into 3 sub-characteristics: Time Behaviour, Resource Behaviour and Efficiency Compliance [FHRF12].

The characteristics time behaviour and performance complying to efficiency were already described in this chapter. A key feature of an automatic test is the efficient evaluation and reporting of the results. How fast can bugs in the software be detected? How significant is the test result related to the quality of the software? In the following sections I would like to answer these questions.

4.4.1 Evaluation of test results

When a test fails which actually should not fail, this test is repeated manually and the fail is verified. Since the description of the test cases is insufficient, the reason for the failure is difficult to find. Up to 100 different measurement results are evaluated in one test case. If only one measurement result shows a deviation, this item has to be identified and the values have to be checked. If the error is then verified by another person, a new bug entry in the JIRA[2] system is created.

4.4.2 Estimation of the result

Goal of any automatic test is, to make a statement regarding the stability and the quality of the software product. In the case of AVL IndiCom all executed tests have to get a positive result. Because of incompatibilities of the references with different measurement devices, this condition was limited to the hardware device "AVL IndiSet 670". This condition is currently considered as sufficient, however the extension of the automatic test for all devices has to be developed.

[2] JIRA is a proprietary issue tracking product, developed by Atlassian

4.5 Maintainability

The ISO 9126 standard defines maintainability as “a set of attributes that bear on the effort needed to make specified modifications” [BAJ93]. The automatic test is implemented in AVL IndiCom itself, therefore a knowledge of this software and the involved script language is a prerequisite.

4.5.1 Maintenance of test cases

As already described in chapter 3, the execution for each single test is basically the same. The parameters like parametrization, reference file and measurement set-up, can be easily edited with AVL IndiCom. All files concerning the automatic test are administered in a version control system[3].

A more challenging and complex task is the administration of the tolerances. The tolerances are defined in a text file for each test and each measurement result. Since more and more absolute tolerances were needed, the administration with a text file is not practicable any more.

4.5.2 Maintenance of the system

The test system itself consists of a set of macros and scripts implemented in AVL IndiCom. Most of the mathematical functions used for the automatic test are also productive features of the software and are also available in the released version. Implementing new features and extensions to the automatic test is not a complex task, given that the user has knowledge of the functionality of the system. To improve the maintainability, a documentation and design document of the system will be necessary.

4.6 Portability

Portability is defined as “a set of attributes that bear on the ability of software to be transferred from one environment to another”‘[BAJ93]. Primarily the automatic test was developed for a single measurement device and the portability of the system was not considered. The porting of the system to another PC and a new measurement device presumes that all hardware devices are available, connected and installed on the new PC.

4.6.1 Compatibility to different measurement devices

If the hardware set-up is supplied, the normal initialization of the test environment has to be done like described in chapter 3. The commissioning of the automatic test

[3] CA Harvest software change manager, Version Control

then is completed. The problem when porting the automatic test to a new hardware, are the references and the organization of the test cases. An administration in the version control system is possible but not recommended, since the maintenance of the references and tolerances will increase.

4.6.2 Installation of the automatic test

The installation routine for the automatic test consists mainly of downloading the work environment of the automatic test from the version control system into the working folder of AVL IndiCom. Then this work environment folder has to be loaded into AVL IndiCom and the test environment is ready to use.

Chapter 5

Robustness of the signal evaluation

This chapter will emphasize testing the comparison method used for the test evaluation against signal distortion. The chosen method for the comparison of the reference signal to measured data was never tested against incompatible datasets. Environmental influences on the measurement are hard to simulate in a real environment and they are not predictable. Also, a lack of simulation files impedes the investigation of the statistical analysis.

Different measurement devices vary in their precision of measurement as well as in their functionality. Generating references for each single hardware device will increase the effort for the maintenance considerably. Therefore, an analysis method has to be found, which is robust against small deviations from measurement results.

5.1 Gaussian White Noise

A common method of simulating measurement influences is a white noise signal having a Gaussian PDF[1]. Such a signal has a relatively flat signal spectrum density. White Gaussian noise generators can serve as useful test tools in solving engineering problems. Test and calibration of communication and electronic systems, cryptography and RADAR jamming are examples of noise generator applications. White noise contains all frequencies in equal proportion and therefore is a convenient signal for system measurements, and experimental design work. Furthermore, noise generators are used in a variety of testing, calibration and alignment applications especially with radio receivers. Consequently, white noise sources with calibrated power density have become standard laboratory instruments. A few of the parameters that can be measured with these sources are: Noise Equivalent Bandwidth, Amplitude Response and Impulse Response [AMKS08].

[1] Probability Density Function

$$n(x) = \sigma * \Phi \tag{5.1}$$

$$f(x) = f(x)_{ref} + n(x) \tag{5.2}$$

Equation (5.1) is the definition of the white Gaussian noise which will be added to distort the signal, where σ is the deviation factor and Φ is a set of normally distributed random numbers[2] with the size of the reference signal. The equation shown in equation (5.2) defines the simulated measured signal by adding the noise vector.

5.2 Experiments

Cycle based measurement results are considered as very similar. Hence, a cylinder pressure curve measured on a gasoline engine was chosen. Measurements on other types of engines like diesel engines or even formula 1 engines do not differ considerably from this curve progression.

For the simulation environment the multi-paradigm numerical computing environment MATLAB was chosen. In the first experiment I will add a white Gaussian noise in the range of D.

$$D = [0.01; 10] \tag{5.3}$$

and with a step size of $\Delta\sigma = 0.01$, for a set over 500 samples. This means, that in this range for each σ, 500 random noise samples will be generated and each sample is compared with the reference signal. The evaluation will then show, how often the evaluation results in a violation of the +/- 1% deviation of the compared value described in chapter 3.

[2] Probability Density Function

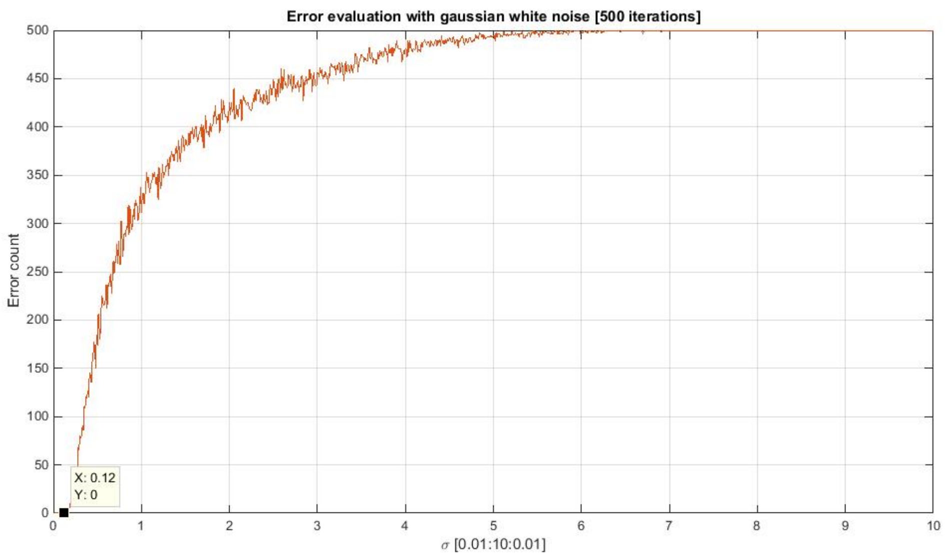


Figure 5.1: *Evaluation for error tolerance*

Figure 5.1 shows the result of the test run. The report shows, that a σ exceeding a value of approximately 0.12 will result in an error. In the following see how the single statistical values behave.

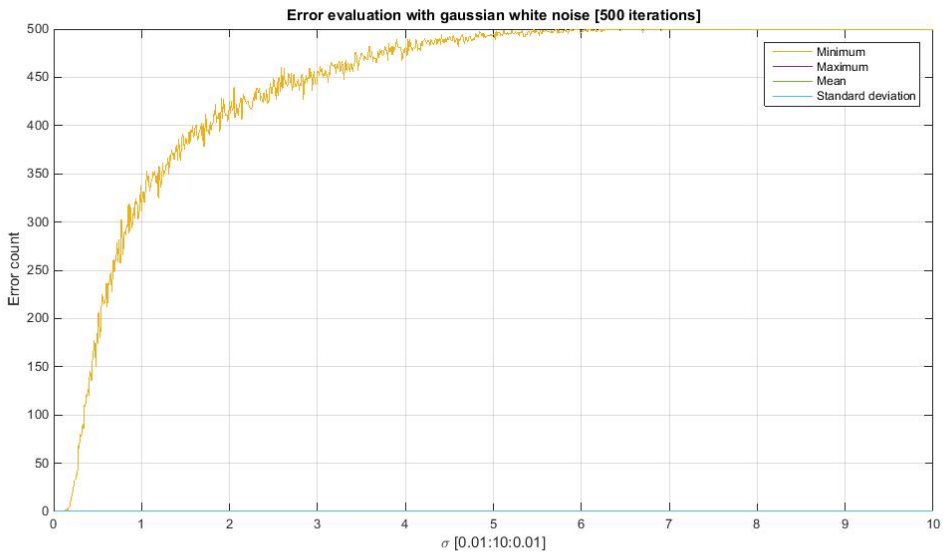


Figure 5.2:

$$Min,\, Max,\, \overline{x}\, and\, \sigma_x\, for\, [\sigma >= 0.01; \sigma <= 10]$$

Figure 5.2 shows, that the minimum value is very sensible to interference of the signal. This is obvious since 1 % of a very small value result in an even smaller value. This figure also indicates, that all other statistical values are not relevant for a small σ, since they do not produce any errors in this range. To analyse the behaviour of the other values, the range of σ will be extended from 10 to 100 with $\Delta\sigma = 0.1$.

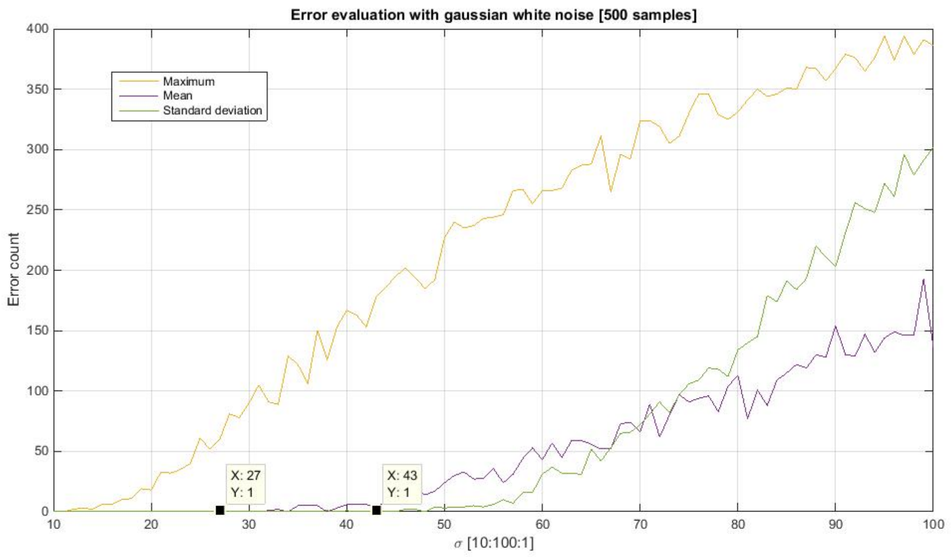


Figure 5.3:

$$Min,\ Max,\ \overline{x}\, and\, \sigma_x\, for\, [\sigma >= 10; \sigma <= 100]$$

Figure 5.3 shows the trend of Min, Max, $\overline{x}$ and the standard deviation σ_x. While the maximum value produces errors for a σ above 10, the threshold for the mean value $\overline{x}$ and the standard deviation σ_x lies above a value of $\sigma = 27$.

An examination of the produced defective data sets showed, that a noise level above a $\sigma >= 6$ is definitely to be considered as an error. Higher values show considerable differences between the two signals.

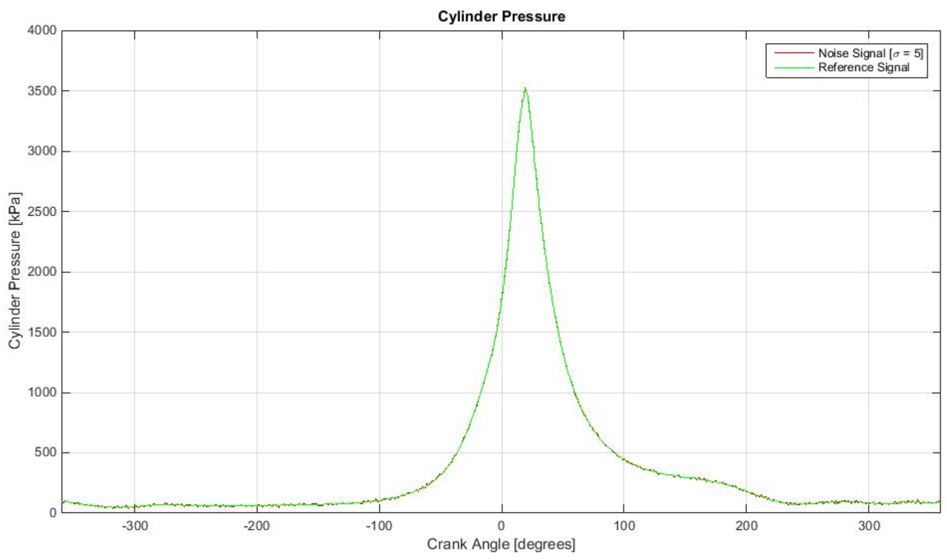


Figure 5.4: *Cylinder pressure with noise* $\sigma = 5$

Figure 5.4 shows a reference cylinder pressure chart and a chart with Gaussian White Noise with $\sigma = 5$. Noise levels beneath this level can be considered as valid, whereas noise levels beyond this level show heavy interferences as shown in figure 5.5.

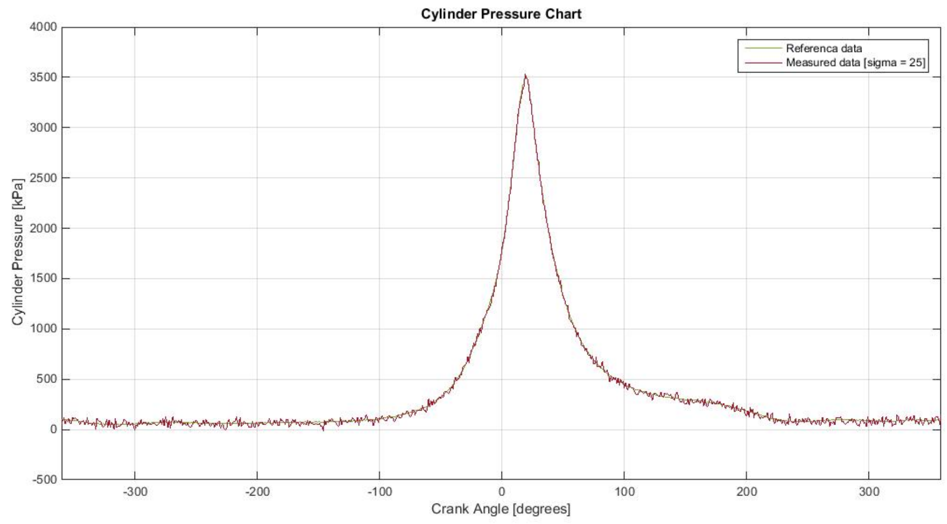


Figure 5.5: *Cylinder pressure with noise* $\sigma = 25$

5.3 Conclusion

The experiments in this chapter showed, that the current comparison method is not appropriate for the evaluation of engine indication data. The values for Maximum and Minimum depend on the value range of the signal and do not include the measurement precision of the connected hardware. As shown in figure 5.3 the mean value and the standard deviation are resistant too low noise levels but too insensitive to detect measurement failures. It may be inferred, that the values for $\overline{x}$ and σ_x are not capable for error detection, since the minimum value will always influence the result.

As already shown in chapter 4.1.1, this method is totally inadequate for other signal types. Data sets with very small values cannot be evaluated with this method, because it shows no link between the reference and the compared signal.

Chapter 6

Improvements

In chapter 4, some serious problems of the current automatic test system were identified. These knowledge lad to examine other procedures for signal comparison and to evaluate some proceedings which could improve the automatic test system. In this chapter I will provide some implementations and tools, which will improve the stability and performance of the testing system.

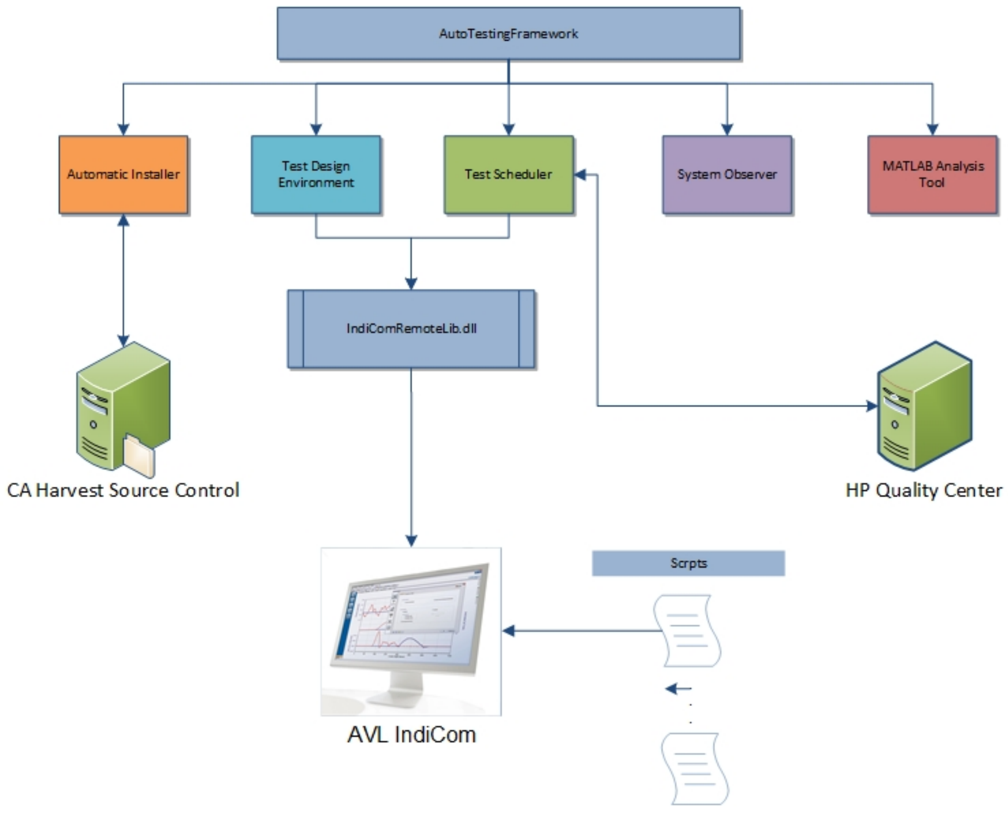


Figure 6.1: *Structure chart of the Automatic Test Framework*

Figure 6.1 shows the structure of the new framework. The automatic installer tool will permanently check for new AVL IndiCom builds and then automatically install the new application and download all resources required to execute the automatic test. This tool will improve the maintainability and usability considerably. For the creation of new test cases, programming skills were needed and this was a reason why this task was neglected. Therefore, a test design environment to ease this task was created. The system observer tool will measure the performance of memory and time consumption. The execution of different test plans for different measurement devices was not supported by the automatic test system and the evaluation of the results was confusing and often not informative. This problem could be solved by integrating the automatic test cases into "HP Quality Center"[1]. The execution of the prepared test plans will be performed by the test scheduler. Failed test cases were analysed and compared with an older version of AVL IndiCom. This is sufficient to detect version incompatibilities and for regression testing, but for the creation of new test cases there was no proof of the references and comparison results. For this reason a analysis tool was developed in MATLAB.

The following sections I will emphasize the tools and improvements based on ISO-9126 standards from chapter 4.

6.1 Functionality

The main requirements to the automatic test system are locating inconsistencies in data acquisition, detecting version conflicts in the software and performances losses. Almost all requirements could not be satisfied or were only partially implemented. The enhancement of these requirements was a mandatory task for this thesis.

6.1.1 Signal evaluation

The unsuitability of the current comparison method was shown in chapter 3. In this section an alternative method will be introduced, which will not only reduce the "false positives" and "false negatives" detections but also will be applicable to different measurement types.

Pearson correlation coefficient

> The Pearson product-moment correlation coefficient is a dimensionless index, which is invariant to linear transformations of either variable. Pearson[2] first developed the mathematical formula for this important

[1] HP Quality Center is a quality management software offered from HP Software Division of Hewlett-Packard

[2] Karl Pearson (27 March 1857, London - 27 April 1936, Coldharbour, Surrey), British statistician, leading founder of the modern field of statistics

measure in 1895:

$$r = \frac{\sum(X_i - \overline{X})(Y_i - \overline{Y})}{[\sum(X_i - \overline{X})^2 \sum(Y_i - \overline{Y})^2]^{\frac{1}{2}}} \tag{6.1}$$

This, or some simple algebraic variant, is the usual formula found in introductory statistics textbooks. In the numerator, the raw scores are centred by subtracting out the mean of each variable, and the sum of cross-products of the centred variables is accumulated. The denominator adjusts the scales of the variables to have equal units. Thus Equation 6.1 describes r as the centred and standardized sum of cross-product of two variables. Using the Cauchy-Schwartz inequality, it can be shown that the absolute value of the numerator is less than or equal to the denominator [e.g. LBN08, Lord and Novic 1968, p. 87]; therefore, the limits of t 1 are established for r. Several simple algebraic transformations of this formula can be used for computational purposes. [LRN88].

With regard to the problem definition, we define x as the reference data set and y as the measured data set.

$$r = \frac{s_{xy}}{s_x * s_y} = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2 \sum_{i=1}^{n}(y_i - \overline{y})^2}} \tag{6.2}$$

In equation 6.2 s_{xy} is the empirical covariance, s_x and s_y are the empirical standard deviations, $\overline{x}$ and $\overline{y}$ are the mean values of the two signals and n is the amount of data point pairs (x_i, y_i).

If the correlation coefficient $r > 0$, a positive relation exists and if $r < 0$ a negative relation exists. There is no linear relation if r = 0. The correlation coefficient always lies between -1 and +1. If r is close to 0 there is only a weak linear connection between the data sets. There are no consistent rules for the evaluation of the correlation coefficient, but the following definition is often found in the literature:

$0.0 <= r <= 0.2$	=>	no to weak linear connection
$0.2 < r <= 0.5$	=>	weak to moderate linear connection
$0.5 < r <= 0.7$	=>	significant linear connection
$0.8 < r <= 1.0$	=>	high to perfect linear connection

Table 6.1: Interpretation of the correlation coefficient r

To test the robustness of this statistical value on indication data, the same experiments as in the previous chapters (5) will be executed for the correlation coefficient r.

First, signals with a Gaussian white noise in the range of D will be examined,

$$D = [0.01; 10] \tag{6.3}$$

with a step size of $\Delta\sigma = 0.01$ for a set of 500 randomly generated samples.

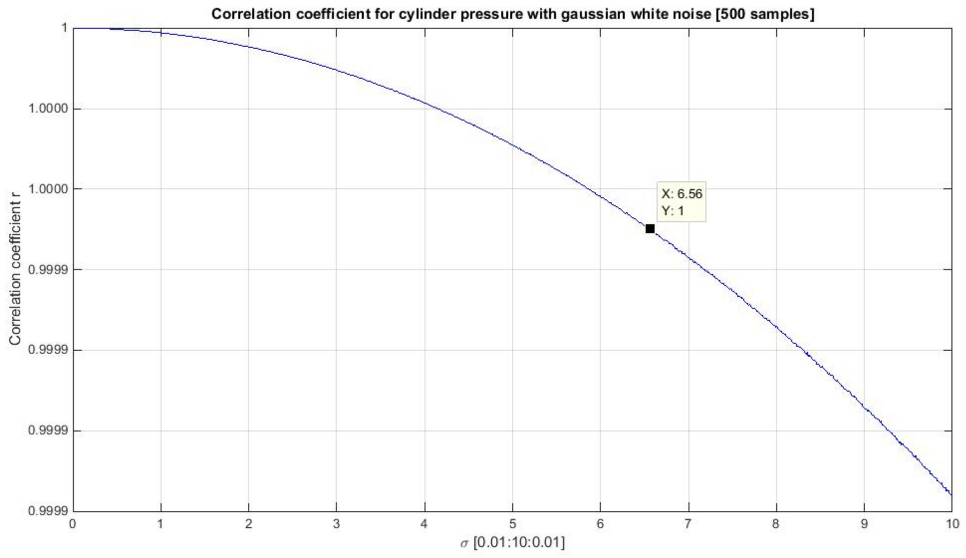


Figure 6.2: *Evaluation of r in the range of $\sigma = [0.01; 10]$*

Figure 6.2 shows the result of the test run. The report shows that a σ exceeding a value of approximately 6.56 is the threshold for a perfect linear connection. The sign of r is not relevant for the evaluation, because we are not interested in the type of linear relationship. Therefore we assume $r = abs(r)$. This experiment shows that the correlation coefficient is robust against noise for a small σ.

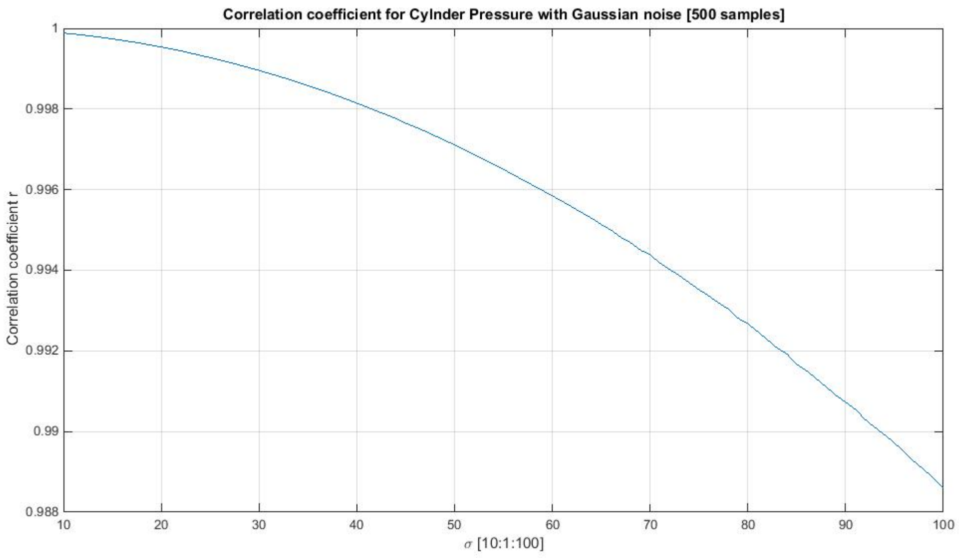


Figure 6.3: *Evaluation of r in the range of $\sigma = [10; 100]$*

Figure 6.3 demonstrates that r is insensitive against noise even for a very high σ. For the automatic test this implies, that a strong correlation with $abs(r) > 0.999$ will be a good evidence for the similarity of the two data sets.

The correlation coefficient seems to fit perfect for the comparison of the two data sets, but there is one obstacle. The correlation coefficient is invariant against translations an scaling of the values, it only provides information about the linear connection of the two signals. The translational displacement can be ignored for cycle based data because the x range will always be the same $[-360° : +360°]$. This is ideal for time based measurements, since they will always have a time delay like described in section 3.3.2.

To avoid errors referring to the scale, a tolerance area has to be defined. For the calculation the signal range of the reference signal will be used.

$$R_{ref} = Max(f_{ref}) - Min(f_{ref}) \tag{6.4}$$

$$T = (R_{ref}/1000) * \mu \tag{6.5}$$

$$t_{upper} = Max(f_{ref}) + T \tag{6.6}$$

$$t_{lower} = Min(f_{ref}) - T \tag{6.7}$$

Equation 6.4 calculates the range of the signal on behalf of its spatial expansion (R_{ref}) . In equation 6.5 the tolerance is calculated on behalf of the range, where μ

is a scale factor. A set of tests showed, that 3 per thousand is an adequate value for μ.

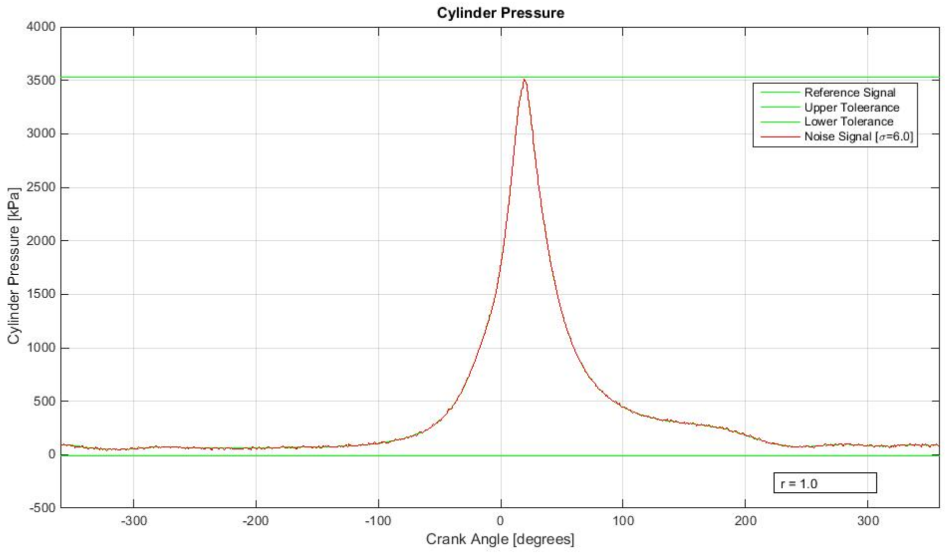


Figure 6.4: *Reference signal and compare signal* $[\sigma = 6.0, \mu = 3]$

A test of time-based measurements shows the same results.

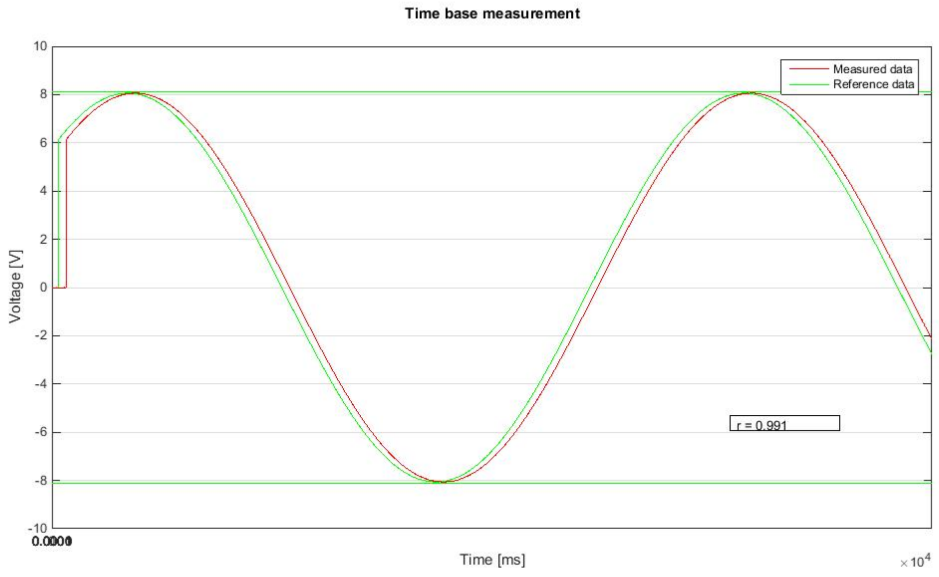


Figure 6.5: *Time-based signals with tolerance area* $[\mu = 3]$

Figure 6.5 shows the evaluation of time based signals and a time delay of 2.995 seconds. The results demonstrate, that the correlation coefficient does not change considerably since r = 0.991. The translational displacement influences the linear connection because of the 0 values at the beginning of the measurement, but the correlation is still considered as perfect. It is assumed that for a time-based measurement a correlation $r > 0.99$ is sufficient.

Finally, the correlation behaviour is checked against totally different signals, which passed the former comparison method like shown in section 4.1.1.

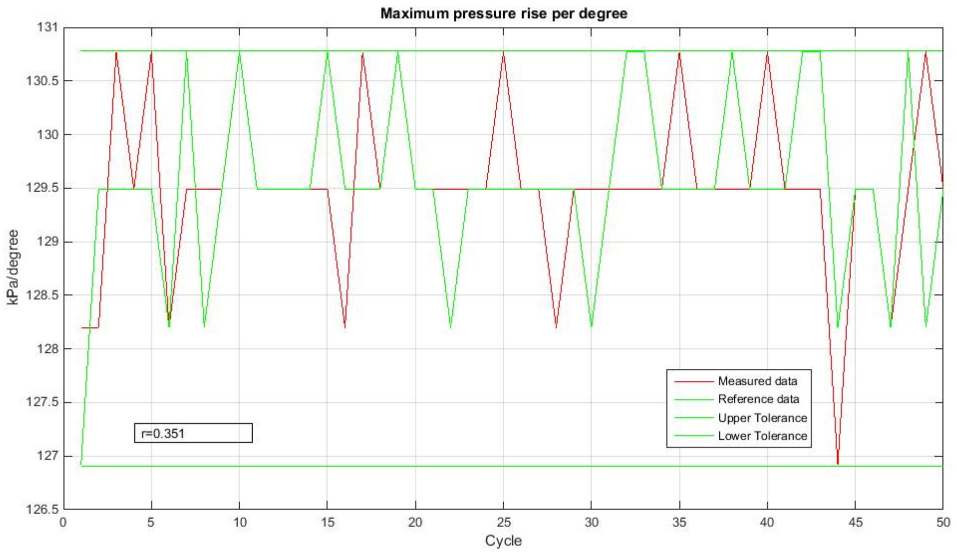


Figure 6.6: *Bad correlated signals* [$\mu = 3$]

Figure 6.6 shows that the correlation coefficient changed considerably under the threshold of 0.999 and now this test wil fail.

The experiments in this section show, that the Pearson Correlation Coefficient is an adequate measurement for the similarity of indication signals. The tolerances can be reduced to an absolute value for r and the scale factor μ for the tolerance area. Also time-based signals can be evaluated with this method and the same tolerance parameters.

This comparison method was implemented as an AVL IndiCom script and integrated into the automatic test system. The effort was manageable and the calculation of r does not change the runtime, since r can also be calculated in $O(n)$.

6.1.2 Performance analysis

Memory profiling and run time analysis are important tasks during automatic testing. No other testing method provides the possibility of tracking the usage of particular instructions in the program code. Profiling not only serves to detect program failures, it also serves to aid program optimization.

As shown in chapter 4, the current performance values obtained from the automatic test are not sufficient. The requirements to the automatic test demand a tracing of the memory consumption and the runtime. The development of new measurement devices and interfaces requires also a tracking of the network load as well as the utilization of the CPU kernels.

Basically, there are two methods for how these performance parameters can be obtained. One possibility is to include the profiling code into the source code itself. Alternatively there is a large number of profiling tools available. The automatic test is a program instance of its own and should test the functionality of the software without interfering with the development.

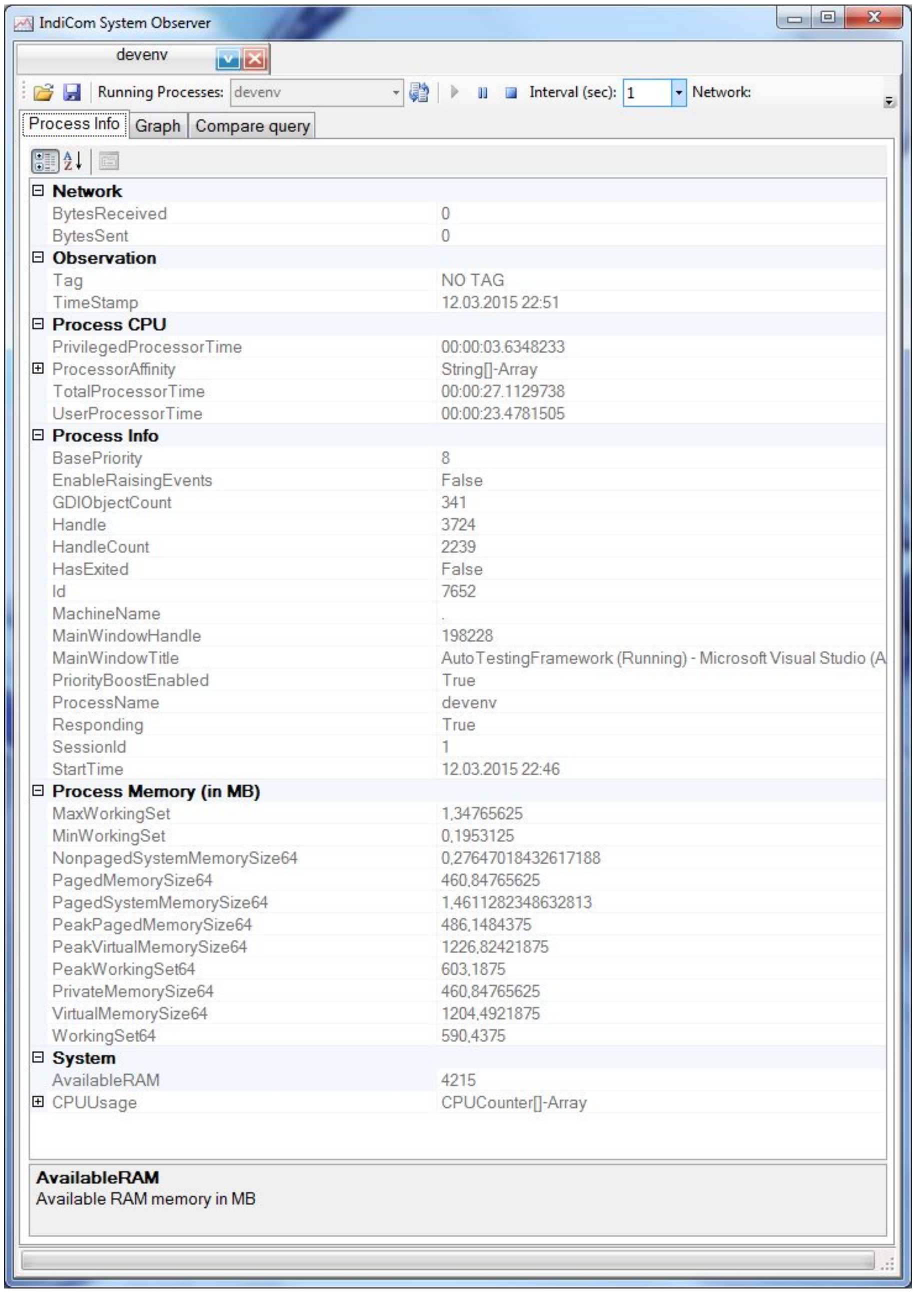
> Including the profiling code into the program bears the risk of errors in the final product. Therefore, the act of inserting profiling codes into a program can slightly modify its behaviour. Thus, in implementing a profile checker, the profiler can avoid the program being profiled to help identify profiling errors. The checker observes the runtime behaviour of a program the same way a human uses a debugger to debug a program: by single stepping, setting breakpoints, running until breakpoints are reached, and examining the memory space of the program. In the process, the checker counts the profiling events as they appear and checks whether the resulting counts match those generated by the profiler. The checker also controls the execution of the original program on the original input file and takes the event counts data as an input and produces diagnostics and verification output as appropriate. [see OB10, p. 123-125]

Different profiling tools were evaluated, but most of them were not suitable for the automatic test, while some were to expensive and others exceeded the requirements and were too complicated. Therefore the decision in favour was to implement a tool that suited the requirements of the automatic test.

AVL IndiCom is developed exclusively for Windows platforms. So the implementation of this tool in C# was obvious. The "System.Diagnostics" name space in the .NET Framework 4.5 provides classes that allow you to interact with system processes, event logs, and performance counters. With the "Process" class it is possible, to monitor system processes across the network and to start and stop local system processes. It is possible to retrieve lists of running processes as well as the current CPU usage. An instance of this class is used to monitor the AVL IndiCom process[3]. With the "PerformanceCounter" class it is possible to monitor the system and process performance. [Mic]

[3] Conc32.exe

For this reason a tool called the “System Observer” was implemented. This tool runs as an instance of its own and is executed in parallel to the automatic test. When the AVL IndiCom process is started, it records the process information on a certain time interval. It also can receive triggers and can then record the information at a specific moment.



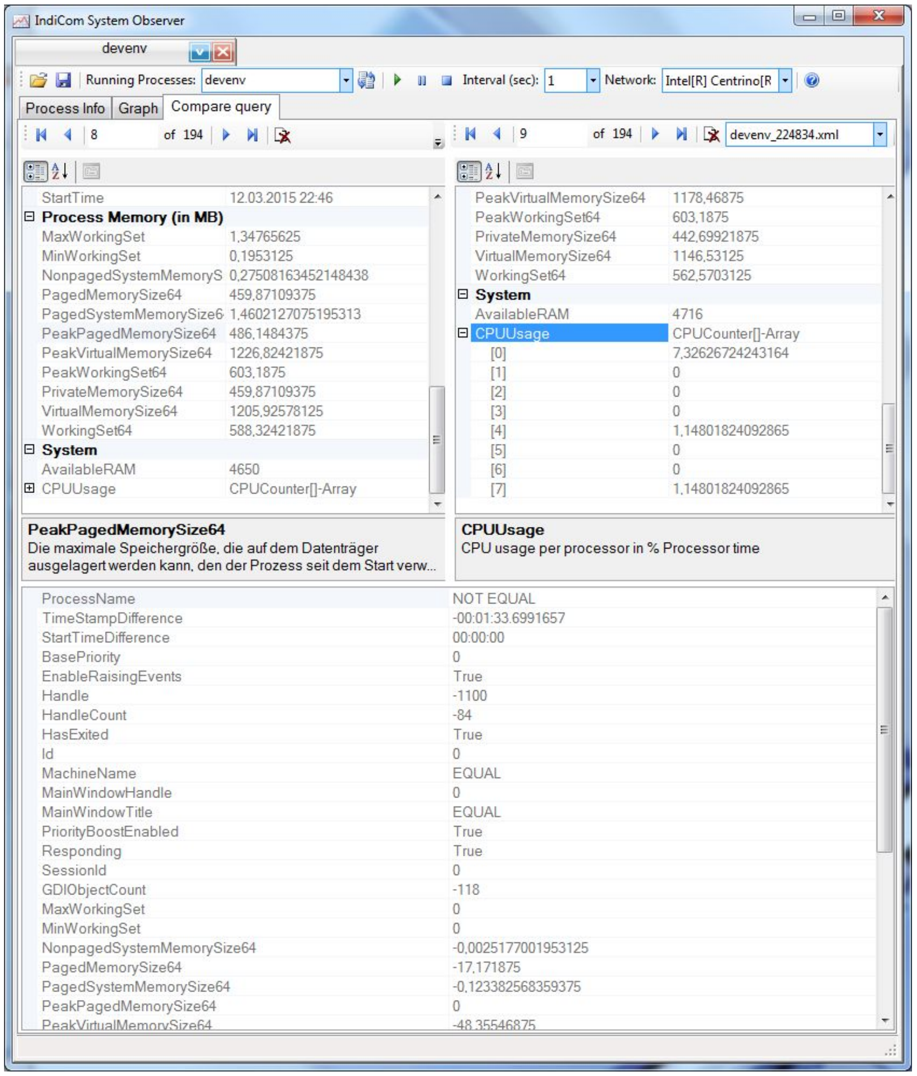
Network	
BytesReceived	0
BytesSent	0
Observation	
Tag	NO TAG
TimeStamp	12.03.2015 22:51
Process CPU	
PrivilegedProcessorTime	00:00:03.6348233
ProcessorAffinity	String[]-Array
TotalProcessorTime	00:00:27.1129738
UserProcessorTime	00:00:23.4781505
Process Info	
BasePriority	8
EnableRaisingEvents	False
GDIObjectCount	341
Handle	3724
HandleCount	2239
HasExited	False
Id	7652
MachineName	.
MainWindowHandle	198228
MainWindowTitle	AutoTestingFramework (Running) - Microsoft Visual Studio (A
PriorityBoostEnabled	True
ProcessName	devenv
Responding	True
SessionId	1
StartTime	12.03.2015 22:46
Process Memory (in MB)	
MaxWorkingSet	1,34765625
MinWorkingSet	0,1953125
NonpagedSystemMemorySize64	0,27647018432617188
PagedMemorySize64	460,84765625
PagedSystemMemorySize64	1,4611282348632813
PeakPagedMemorySize64	486,1484375
PeakVirtualMemorySize64	1226,82421875
PeakWorkingSet64	603,1875
PrivateMemorySize64	460,84765625
VirtualMemorySize64	1204,4921875
WorkingSet64	590,4375
System	
AvailableRAM	4215
CPUUsage	CPUCounter[]-Array

AvailableRAM
Available RAM memory in MB

Figure 6.7: *The “System Observer” configuration*

In figure 6.7 the configuration page of the profiling tool is shown. In the menu bar the process to be observed can be selected, the time interval for automatic observation can be set and the network interface which should be tracked can be selected. A manual trigger can simply be made by calling the executable file over the command line. By passing a text as parameter over the command line, the profiling record can be tagged with this text. Before and after the execution of an automatic test, the system observer is triggered and the current process profiling information is saved. This is realized by simply serializing the process information into a file.

In figure 6.7 all recorded process parameters can be seen. The widespread of the captured values ranges from network load to CPU load and memory consumption. A specification of the single values is shown by selecting the value, therefore this will not be emphasized. Each measurement series can be saved and loaded for comparison later on.



StartTime	12.03.2015 22:46
Process Memory (in MB)	
MaxWorkingSet	1,34765625
MinWorkingSet	0,1953125
NonpagedSystemMemoryS	0,27508163452148438
PagedMemorySize64	459,87109375
PagedSystemMemorySize6	1,4602127075195313
PeakPagedMemorySize64	486,1484375
PeakVirtualMemorySize64	1226,82421875
PeakWorkingSet64	603,1875
PrivateMemorySize64	459,87109375
VirtualMemorySize64	1205,92578125
WorkingSet64	588,32421875
System	
AvailableRAM	4650
CPUUsage	CPUCounter[]-Array

PeakPagedMemorySize64
Die maximale Speichergröße, die auf dem Datenträger ausgelagert werden kann, den der Prozess seit dem Start verw...

PeakVirtualMemorySize64	1178,46875
PeakWorkingSet64	603,1875
PrivateMemorySize64	442,69921875
VirtualMemorySize64	1146,53125
WorkingSet64	562,5703125
System	
AvailableRAM	4716
CPUUsage	CPUCounter[]-Array
[0]	7,32626724243164
[1]	0
[2]	0
[3]	0
[4]	1,14801824092865
[5]	0
[6]	0
[7]	1,14801824092865

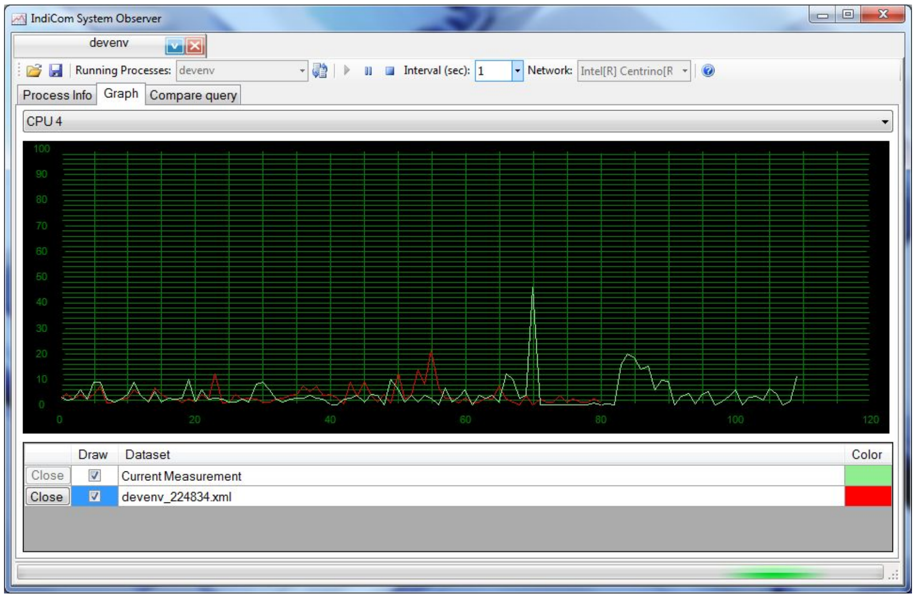
CPUUsage
CPU usage per processor in % Processor time

ProcessName	NOT EQUAL
TimeStampDifference	-00:01:33.6991657
StartTimeDifference	00:00:00
BasePriority	0
EnableRaisingEvents	True
Handle	-1100
HandleCount	-84
HasExited	True
Id	0
MachineName	EQUAL
MainWindowHandle	0
MainWindowTitle	EQUAL
PriorityBoostEnabled	True
Responding	True
SessionId	0
GDIObjectCount	-118
MaxWorkingSet	0
MinWorkingSet	0
NonpagedSystemMemorySize64	-0,0025177001953125
PagedMemorySize64	-17,171875
PagedSystemMemorySize64	-0,123382568359375
PeakPagedMemorySize64	0
PeakVirtualMemorySize64	-48,35546875

Figure 6.8: *Comparison of the profiling results*

The collection of performance data is useless, if the obtained data is not analysed. The "System Observer" provides the possibility to surf through the collected data and to compare data sets with each other. Figure 6.8 shows the evaluation page. The two top frames provide the possibility to step through the current measurement or to compare the current measurement to a previously captured reference data set. The bottom frame displays the calculated Δ results between the two data sets.

To ease the comparison between two data sets, a graphical display was implemented. There, the single profiling values can be displayed in a chart. In figure 6.9 the comparison of a reference data set with a current performance measurement is shown. Any number of data sets can be loaded and compared.



	Draw	Dataset	Color
Close	✓	Current Measurement	🟩
Close	✓	devenv_224834.xml	🟥

Figure 6.9: *Performance chart [Processor time on kernel 4]*

6.2 Reliability

The term software reliability was already defined as the probability, that the program will work according to its requirements and for a given amount of time [see Pha00, chap. 1]. In chapter 4 the problem of frequent program crashes was described, which influences the result of the automatic test and extends the runtime considerably. Since the automatic test routine is implemented in the software to be tested, this was a challenging task.

6.2.1 Endurance test

To estimate the reliability of the current development build, the automatic test should deliver its results constantly. For this reason a method to remote control

AVL IndiCom had to be found.

In practice, combustion analysis is not a task which is executed separately. On an engine test bench, many different engine parameters are measured and analysed. This means, that a series of different hardware devices and software programs have to communicate and exchange data with another. The main test bench application of AVL is the "AVL PUMA Automation Platform"', a software which provides precise control of measuring devices, test cell facilities, units under test and all safety-relevant monitoring features, in addition to standard features, such as fully automated test runs or manual operation. Since combustion analysis had to be integrated into the automation platform, an interface, the "IndiComRemoteLib.dll" was implemented. This interface provides the basic commands to control a measurement and it also provides the possibility to execute scripts.

All functionality for opening an AVL IndiCom instance, starting and stopping a measurement and the execution of script files. Therefore, the decision to implement the automatic test with this interface was made. The key advantage of this implementation is, that program crashes can be detected and a new instance can then be started. A more detailed description will be found in chapter 6.3.

6.2.2 Test runtime

Because of shorter release cycles of the software, an improvement of the runtime of the automatic test had to be found. The automatic test system was inflexible in changing the test sequence and the administration of the test cases was decentralized. One good approach to reduce the runtime of the automatic test is the application of equivalence class partitioning.

6.2.3 Equivalence class partitioning

> Equivalence class partitioning result in a partitioning of the input domain of the software-under-test. The technique can also be used to partition the output domain, but this is not a common usage. The finite number of partitions or equivalence classes that result allow the tester to select a given member of an equivalence class as a representative of that class. It is assumed that all members of an equivalence class are processed in an equivalent way by the target software.
>
> Using equivalence class partitioning a test value in particular class is equivalent to a test value of any other member of that class. Therefore, if one test case in a particular equivalence class reveals a defect, all the other test cases based on that class would be expected to reveal the same defect. We can also say that if a test case in a given equivalence class did not detect a particular type of defect, then no other test case based on that class would detect the defect (unless a subset of the equivalence

class falls into another equivalence class, since classes may overlap in some cases). [Bur03, chap. 4.5]

As described in chapter 4, the automatic test executes measurements on different simulation files. These simulation files are resumed in different resolutions to verify the accuracy of the results.

Engine	Cylinders	Resolution [°]	Measurement Mode
Gasoline	4	0.1	CA
Gasoline	4	0.2	CA
Gasoline	4	0.5	CA
		...	
Gasoline	4	0.1	RTP
Gasoline	4	0.2	RTP
		...	
Diesel	6	0.5	CA
Diesel	6	1.0	CA
		...	

Table 6.2: Excerpt of the automatic test plan

Table 6.2 shows an excerpt of the current test plan. The same simulation is repeated for different measurement resolutions. Such a set of repetitions can be considered as one equivalence class. The test case with the highest resolution will be used as representative of this class as any other test case of this class would be expected to reveal the same defect.

The definition of equivalence classes leads us to the next improvement of the automatic test, the creation of test plans which will be treated in the next section.

6.3 Usability

The main problems concerning the usability of the automatic test are planning of the test runs, the creation and administration of test cases and the execution of pre defined test plans.

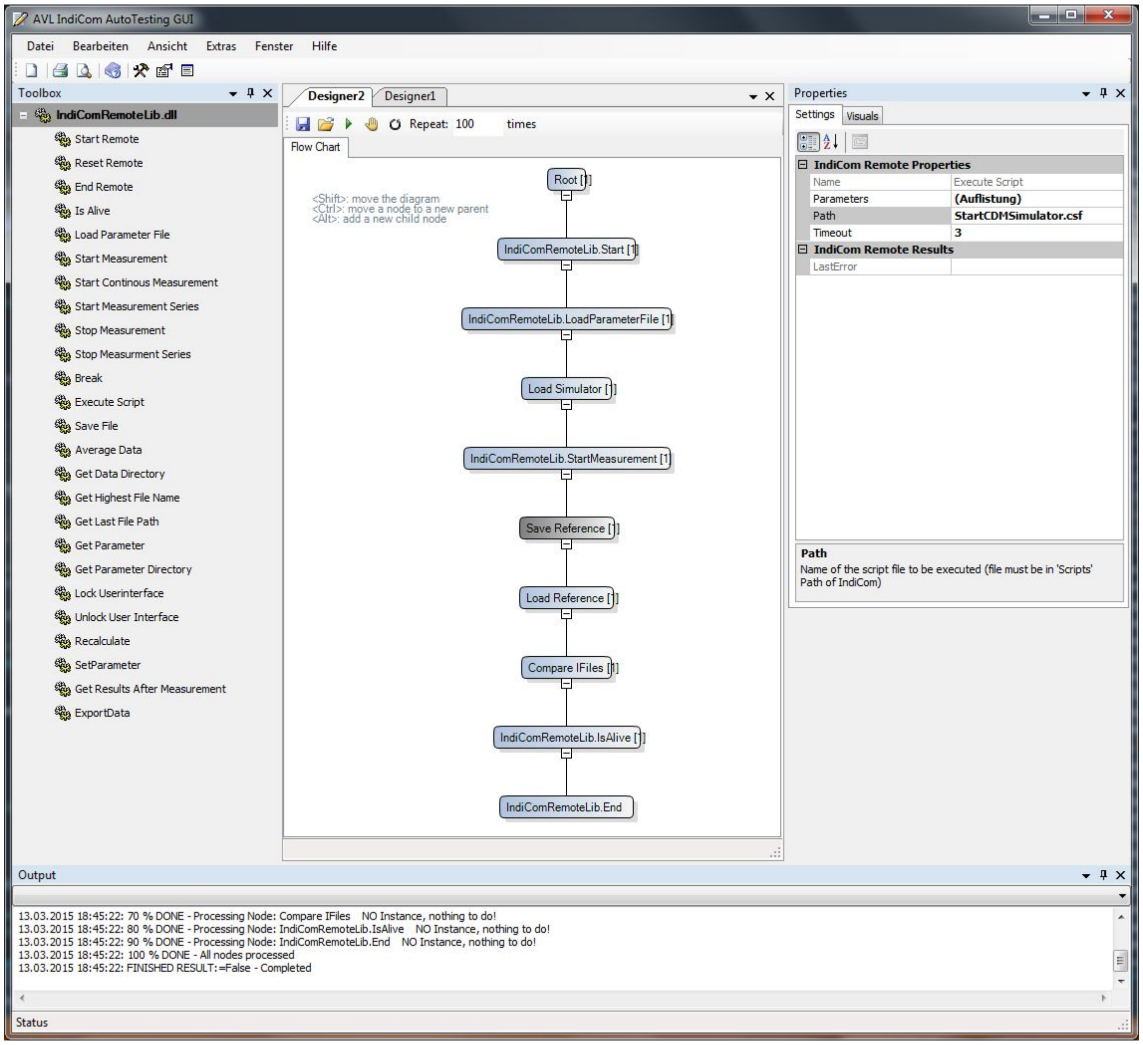
6.3.1 Creation of new tests

As already mentioned, the creation of a new test case required a good knowledge and understanding of the test procedure as well as good programming skills. Since the automatic test should be operable by any member of the AVL IndiCom team, a simplification of that task had to be found.

> Models are used to understand, specify and develop systems in many disciplines. From DNA and gene research to the development of the latest fighter aircraft, models are used to promote understanding and provide a reusable framework for product development. In the software engineering process, models are now accepted as part of a modern object oriented analysis and design approach by all of the major OO methodologies.
>
> Modelling is a very economical means of capturing knowledge about a system and then reusing this knowledge as the system grows. For a testing team, this information is gold; what percentage of a test engineer's task is spent trying to understand what the System Under Test (SUT) should be doing? (Not just is doing.) Once this information is understood, how is it preserved for the next engineer, the next release, or change order? If you are lucky it is in the test plan, but more typically buried in a test script or just lost, waiting to be rediscovered. By constructing a model of a system that defines the systems desired behaviour for specified inputs to it, a team now has a mechanism for a structured analysis of the system. Scenarios are described as a sequence of actions to the system [as it is defined in the model], with the correct responses of the system also being specified. [AD97]

Thus, a model-based test design approach for the automatic test system would be optimal. The AVL IndiCom remote interface constitutes a good basis for the abstraction to such a model. Therefore, the "AutoTesting GUI" was implemented, which simplifies the creation and maintenance of the test cases.

Each remote interface command and each script, contained in the automatic test environment, can be seen as single entity which has input parameters and output parameters. A sequence of such entities then forms a test case. As described in [Bei95] such a model can be defined as Finite State Machine (FSM).



IndiCom Remote Properties	
Name	Execute Script
Parameters	**(Auflistung)**
Path	**StartCDMSimulator.csf**
Timeout	**3**
IndiCom Remote Results	
LastError	

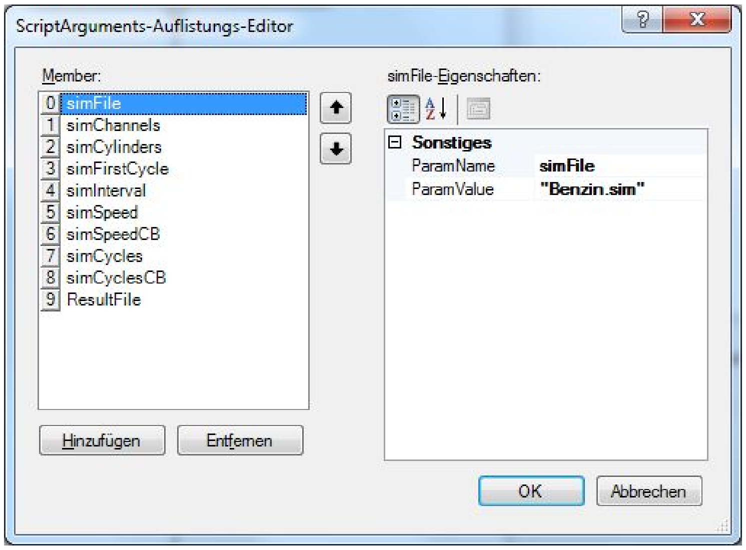
Path
Name of the script file to be executed (file must be in 'Scripts' Path of IndiCom)

```
13.03.2015 18:45:22: 70 % DONE - Processing Node: Compare IFiles    NO Instance, nothing to do!
13.03.2015 18:45:22: 80 % DONE - Processing Node: IndiComRemoteLib.IsAlive    NO Instance, nothing to do!
13.03.2015 18:45:22: 90 % DONE - Processing Node: IndiComRemoteLib.End    NO Instance, nothing to do!
13.03.2015 18:45:22: 100 % DONE - All nodes processed
13.03.2015 18:45:22: FINISHED RESULT:=False - Completed
```

Figure 6.10: *The "AutoTesting GUI" showing a standard test case*

Figure 6.10 shows the prototype of the "AutoTesting GUI". A test case sequence can easily be created by simply dragging the commands from the toolbox (on the right side of the form) to the flow chart window. The sequence will then be executed from top to bottom. Each node in the graph has input parameters and a result. By selecting the node in the graph, its properties will be shown in the properties window, which is located on the left side. By selecting a property, its description will be shown below, which should ease the handling.

The remote interface only consists of a limited set of commands but can be extended by using the "ExecuteScript" command. The "AutoTesting GUI" then automatically checks the input parameters of this script, which can then be edited in the properties window of the "ExecuteScript" node.



Member:

0 simFile
1 simChannels
2 simCylinders
3 simFirstCycle
4 simInterval
5 simSpeed
6 simSpeedCB
7 simCycles
8 simCyclesCB
9 ResultFile

simFile-Eigenschaften:

Sonstiges	
ParamName	simFile
ParamValue	"Benzin.sim"

Figure 6.11: *The "ExecuteScript" parameters for starting the simulator*

Figure 6.11 shows the properties of the "StartCDMSimulator" script, which will be executed before the measurement to initialize the simulator hardware. Each parameter can be changed and adapted to the needs of the current test case.

All nodes have some properties in common. The Name property is also the ID of the node. The time out property defines a maximum runtime for this operation. Once this time has exceeded, this node will produce an error. The "Last Error" property is usually empty, but will show the last error message the respective node has produced. If the node executes successfully, the value is set to empty again. All nodes also posses properties which define the appearance of the node in the graph like colour, caption and size. The "Enable" property defines, whether the node is executed or the transition from its ancestor to the successor is taken.

Of course, the model can be executed in the environment immediately and a counter can be set to define the amount of iterations. It is also possible to set breakpoints on the nodes, so that a test case model can be debugged. Each model then is serialized to an XML file when saved. This eases the handling of the test cases because formerly, the test cases consisted of many different scripts. The output window at the bottom of the window shows the log output for the execution.

This test design environment was developed with different open source code libraries. An extension to conditional statements is planned in the future.

6.3.2 Test plans

Basically, the functionality for creating test plans was available in the former automatic test by removing or adding the test scripts from/to the "Test" folder. The set contained in this folder was then executed in an endless loop in the order of the file names. Therefore it was not possible to execute different test sets one after another as is needed, when working with equivalence classes 6.2.3 and the test order cannot be changed.

Therefore, a decision had to be taken and different options were evaluated. Since AVL introduced the quality management software “HP Quality Center” to the company and all other test methods had to be handled by this software, it was obvious to find a way to manage the automatic test with this tool. “HP Quality Center” can be seen as the industry standard for quality management and provides a wide functionality for managing test plans and reporting figures. Hewlett Packard provides a COM interface for communicating with the central “HP Quality Center” server. Therefore, the test plans can be accessed and the results of the single test cases can be transferred to the quality database.

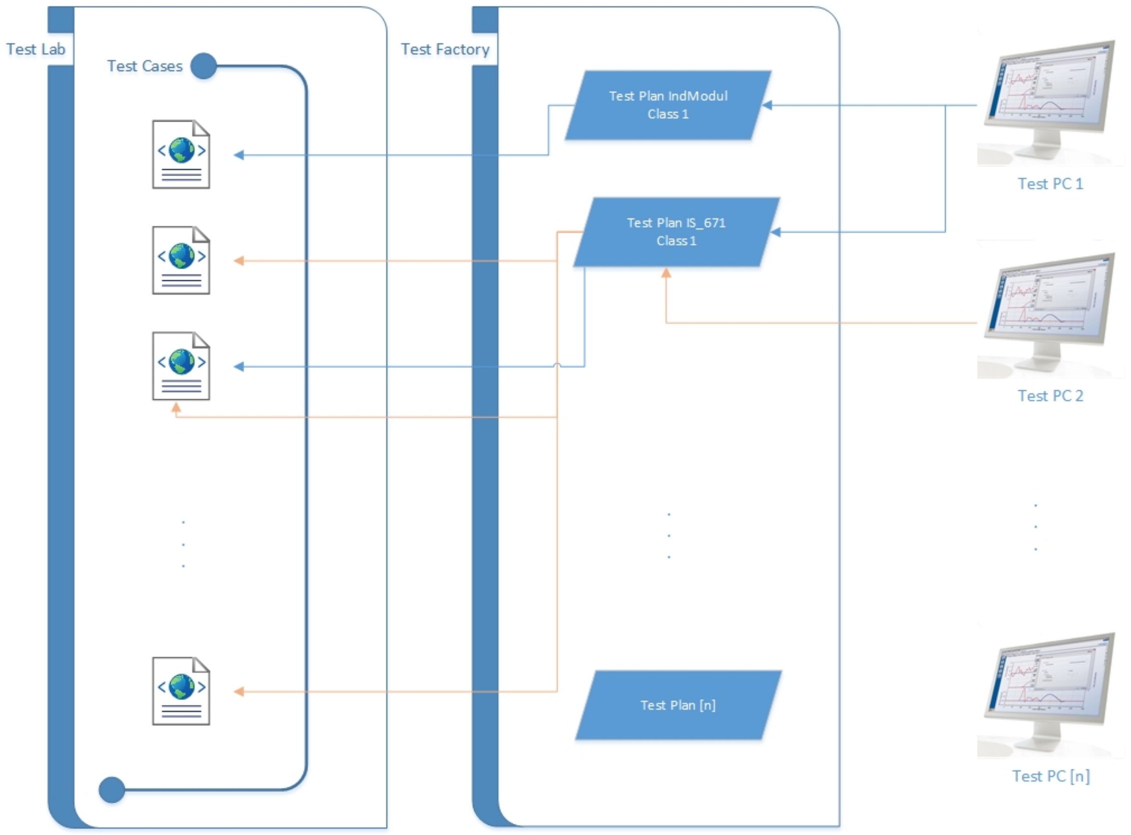


Figure 6.12: *“HP Quality Center” work flow*

Figure 6.12 shows the basic workflow in “HP Quality Center”. In a first step, all test cases are created in the so called “Test Lab”. Here a detailed description for each test case has to be entered and the XML file created with the “AutoTesting GUI” has to be linked as well, so that the case is complete.

In the second step shown in figure 6.12, the test plans are created. Each plan consists of a subset of the test cases from the “Test Lab”. The order of test cases can be arranged as needed. The maintenance of the test plans is intuitive and lots of tools from copying and moving test case from one test plan to another are available. A detailed documentation of these tools can be found in the documentation library inside the “HP Quality Center”.

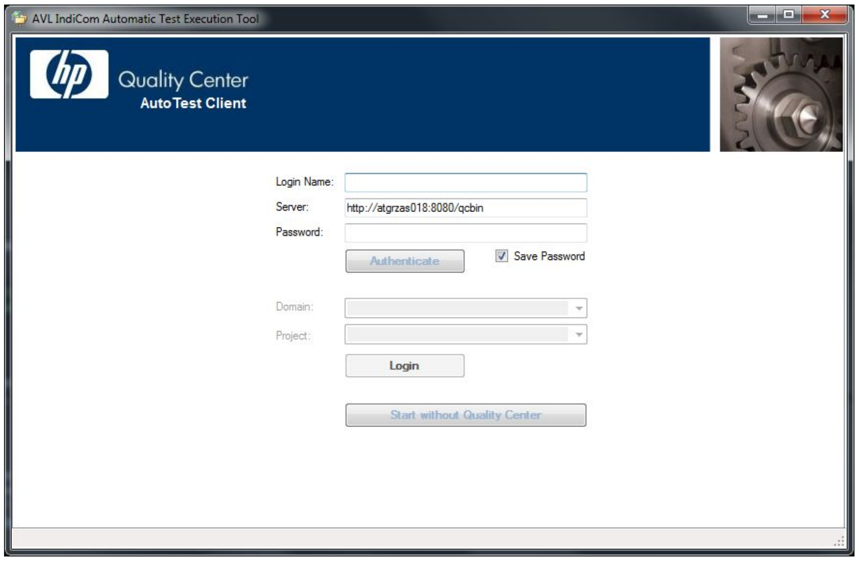
For executing the test plans created in “HP Quality Center” a test execution tool was created, which allows the user to load test plans from the web server and execute them on the local machine. The results will then be sent back to the “HP Quality Center” and then they can be evaluated with the tools provided by the software. A more detailed description of the test execution tool can be found in 6.3.4.

6.3.3 Understandability of test cases

In the former version of the automatic test, the test cases were poorly documented and for people with no programming skills they were difficult to comprehend. This should now change now considerably, since the documentation of each test case in the “HP Quality Center” is mandatory. The “AutomaticTesting GUI” should also contribute to ease the comprehensibility of the work flow of each test.

6.3.4 Execution of test plans

To execute the predefined test plans from “HP Quality Center”, a test execution tool was implemented. To obtain the test plans from the quality center database, an authentication to the “HP Quality Center” has to be made. Figure 6.13 shows the login dialog for the execution tool. If the authentication succeeds, the user can select the test domain and the test project from “HP Quality Center”. By pressing login, the test configuration will be loaded from the quality server. Alternatively the tool can be started in offline mode, where no connection is necessary and the test XML files can be loaded from a local path.

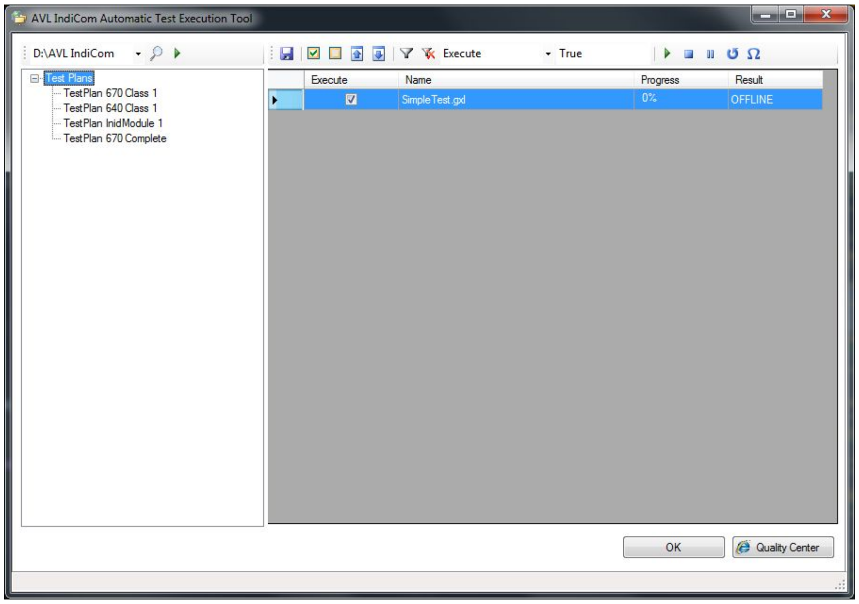

AVL IndiCom Automatic Test Execution Tool
Quality Center
AutoTest Client
Authenticate
Login
Start without Quality Center

Login Name:
Server: http://atgrzas018:8080/qcbin
Password:
Save Password [✓]
Domain:
Project:

Figure 6.13: *"Test Execution Tool" Login dialog*

Figure 6.14 shows the administration page of the "Execution Tool". If logged in to "HP Quality Center", the predefined test plans will be shown in the tree control on the left side of the dialog. The right side will contain the test cases for this plan. The menu bar contains controls for reordering the test cases and filter possibilities. Single tests can be selected by a check box and only checked items will be executed by the tool. On the top left side, the execution controls can be found. The execution can be started, paused and stopped. In addition, options for an endless test and random-order test execution (Ω) are available.

If the tool is connected to the "HP Quality Center", the result of each test case will be sent to the server immediately after execution. Therefore, the current state for every test station can be ovserved through the web interface of "HP Quality Center". The tool aslo has an included browser plug-in and by pressing the "HP Quality Center" button the project homepage of the test project will be shown.



Execute	Name	Progress	Result
✓	SimpleTest.gxd	0%	OFFLINE

Figure 6.14: *"Test Execution Tool" Test plan execution*

6.4 Efficiency

In chapter 4, the problems related to the efficiency of evaluating the test results and the analysis of failed tests were identified and various tools to ease this task are introduced. In this chapter I have already introduced some tools that will ease this task.

6.4.1 Evaluation of test results

The whole evaluation was outsourced to "HP Quality Center", where the sum of all test results from the different work stations can be found. On the local test machines, the "Test Execution Tool" shows the current status and the results of the test cases executed. This represents a great improvement compared to the former test system, where all results were stored in log files on the particular test machine. Now, the test results from the automatic test can be administrated in the same way as for other test modeslike the manual test. At the end of the test phase an overall status of the testing can be generated.

6.4.2 Error Analysis

The "AutoTesting GUI" was already introduced in 6.3.1. With this tool the debugging and execution of single test cases is now possible without any previous knowledge. When executed, the "AutoTesting GUI" will show the error state of each node.

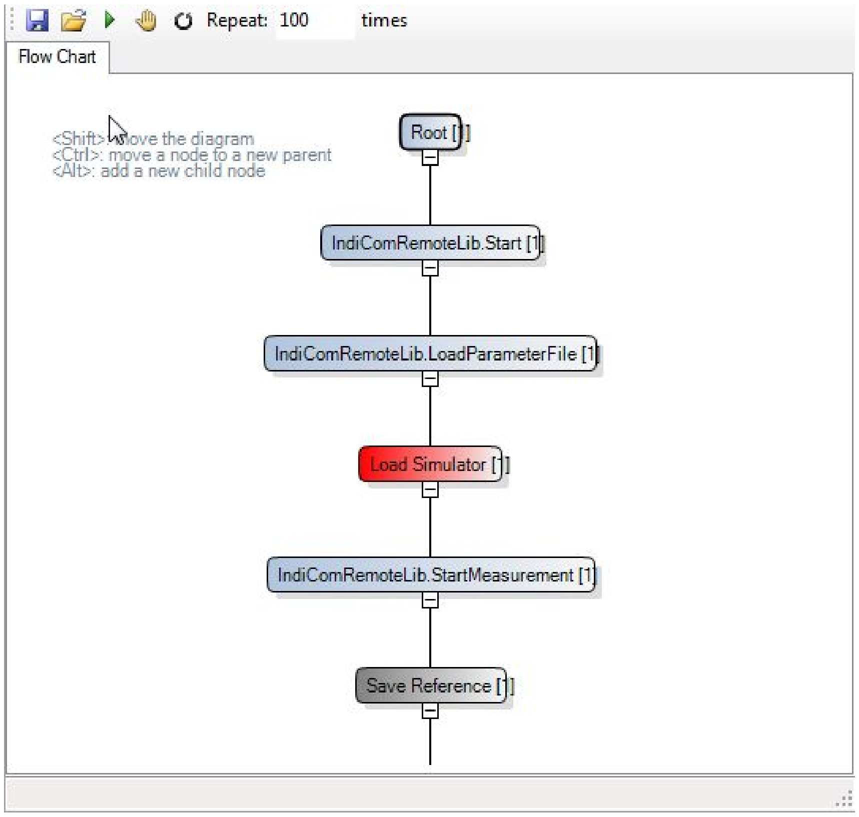
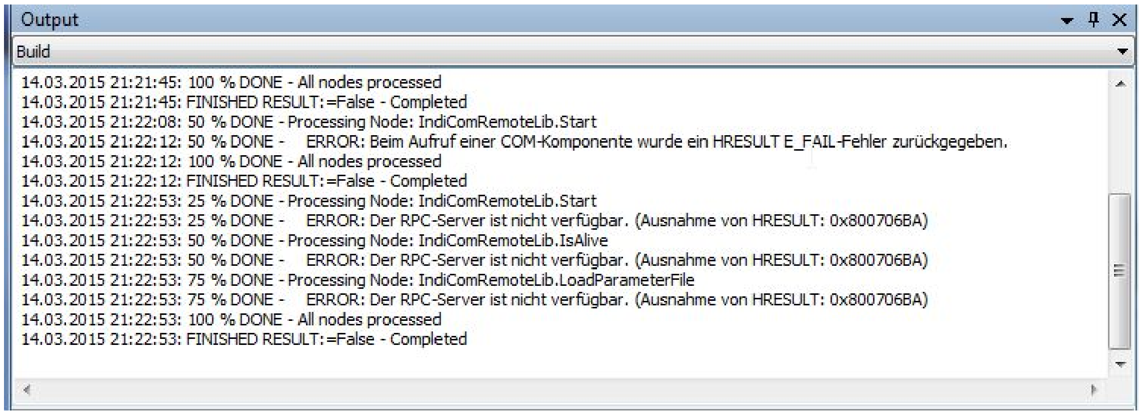

Repeat: 100 times
Flow Chart
<Shift>: move the diagram
<Ctrl>: move a node to a new parent
<Alt>: add a new child node
Root [1]
IndiComRemoteLib.Start [1]
IndiComRemoteLib.LoadParameterFile [1]
Load Simulator [1]
IndiComRemoteLib.StartMeasurement [1]
Save Reference [1]

Figure 6.15: *Test analysis, failed test case*

Figure 6.15 shows the analysis of a failed test. In this case the parameter file that defines the measurement parameters could not be loaded. By selecting the node the error log will be displayed in the output window on the bottom of the "AutoTesting GUI" as you can see in figure 6.16.



```
14.03.2015 21:21:45: 100 % DONE - All nodes processed
14.03.2015 21:21:45: FINISHED RESULT:=False - Completed
14.03.2015 21:22:08: 50 % DONE - Processing Node: IndiComRemoteLib.Start
14.03.2015 21:22:12: 50 % DONE -    ERROR: Beim Aufruf einer COM-Komponente wurde ein HRESULT E_FAIL-Fehler zurückgegeben.
14.03.2015 21:22:12: 100 % DONE - All nodes processed
14.03.2015 21:22:12: FINISHED RESULT:=False - Completed
14.03.2015 21:22:53: 25 % DONE - Processing Node: IndiComRemoteLib.Start
14.03.2015 21:22:53: 25 % DONE -    ERROR: Der RPC-Server ist nicht verfügbar. (Ausnahme von HRESULT: 0x800706BA)
14.03.2015 21:22:53: 50 % DONE - Processing Node: IndiComRemoteLib.IsAlive
14.03.2015 21:22:53: 50 % DONE -    ERROR: Der RPC-Server ist nicht verfügbar. (Ausnahme von HRESULT: 0x800706BA)
14.03.2015 21:22:53: 75 % DONE - Processing Node: IndiComRemoteLib.LoadParameterFile
14.03.2015 21:22:53: 75 % DONE -    ERROR: Der RPC-Server ist nicht verfügbar. (Ausnahme von HRESULT: 0x800706BA)
14.03.2015 21:22:53: 100 % DONE - All nodes processed
14.03.2015 21:22:53: FINISHED RESULT:=False - Completed
```

Figure 6.16: *Log window of "AutoTesting GUI"*

Additionally, a log file containing all results of the automatic test in textual form is stored on the local machine. Therefore, an analysis in case the test could be executed on one machine while failing on another is facilitated.

6.5 Maintainability

As shown in chapter 6.1.1, the effort for maintenance could be reduced because of a new signal evaluation. There is no need of managing a huge data set of tolerances any more.

6.5.1 Maintenance of test cases

As already described extensively in the previous chapters, it was possible to imporve the maintenance of the test cases and test plans considerably due to the application of the "HP Quality Center".

6.5.2 Maintenance of the system

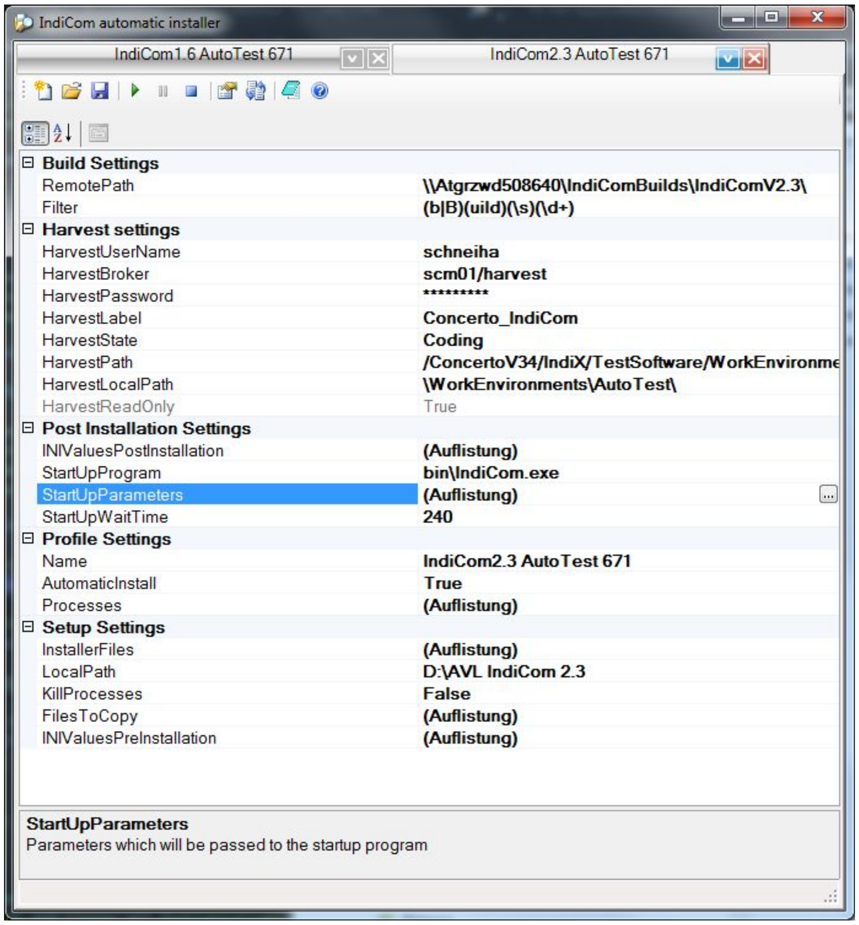
The entire test environment is maintained with the source control software "CA Harvest software change manager" in oreder to prevent data loss. Errors in the test tools cannot be ruled out, but the tools are still in development and will be improved continously. The core functionality of the automatic test is still included in AVL IndiCom as scripts, so that the changes to the system itself were minimal. The developed tools written in C# are maintained by the development team and underlie a continuous quality control through code reviews.

6.6 Portability

The installation routine for the automatic test is a time-consuming task. All the steps described in section 3.2 have to be repeated on each testing work station. Investing in the development of a tool that automates these steps will improve the performance of the automatic test as well as the portability to new hardware devices.

6.6.1 Installation of the automatic test

In order to automate the installation routine, an automatic installer tool was implemented. Basically, this tool executes the steps described in section 3.2. To provide more flexibility a configuration interface was designed.

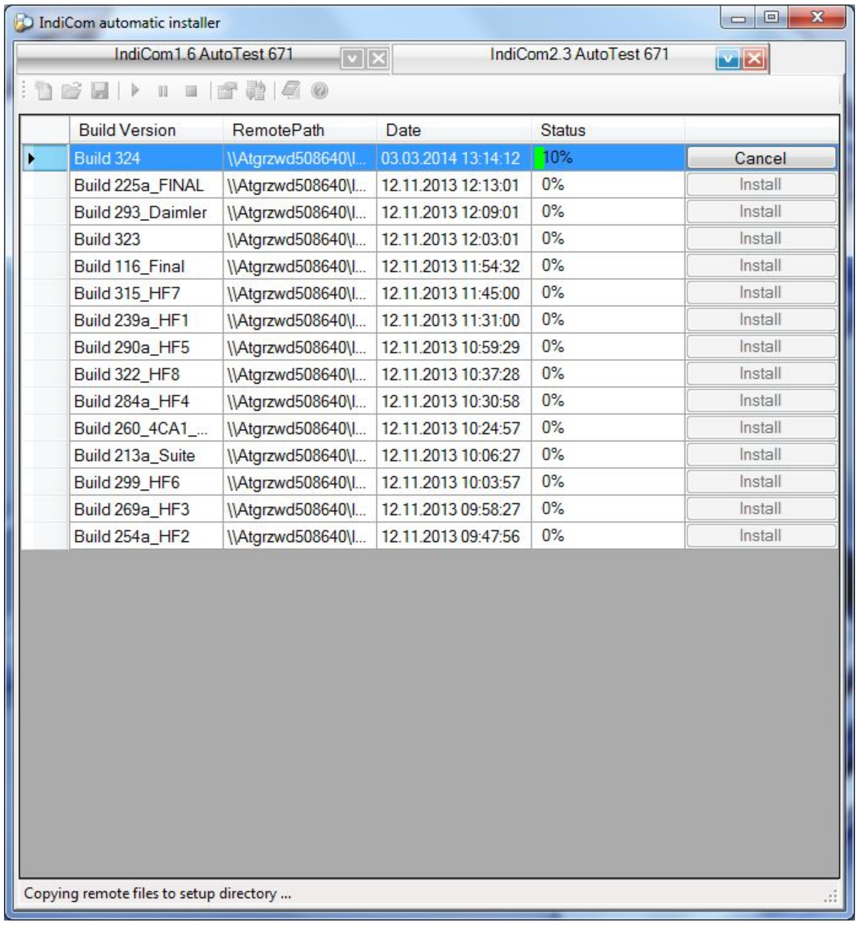


Build Settings	
RemotePath	\\Atgrzwd508640\IndiComBuilds\IndiComV2.3\
Filter	(b|B)(uild)(\s)(\d+)
Harvest settings	
HarvestUserName	schneiha
HarvestBroker	scm01/harvest
HarvestPassword	*********
HarvestLabel	Concerto_IndiCom
HarvestState	Coding
HarvestPath	/ConcertoV34/IndiX/TestSoftware/WorkEnvironme
HarvestLocalPath	\WorkEnvironments\AutoTest\
HarvestReadOnly	True
Post Installation Settings	
INIValuesPostInstallation	(Auflistung)
StartUpProgram	bin\IndiCom.exe
StartUpParameters	(Auflistung)
StartUpWaitTime	240
Profile Settings	
Name	IndiCom2.3 AutoTest 671
AutomaticInstall	True
Processes	(Auflistung)
Setup Settings	
InstallerFiles	(Auflistung)
LocalPath	D:\AVL IndiCom 2.3
KillProcesses	False
FilesToCopy	(Auflistung)
INIValuesPreInstallation	(Auflistung)

StartUpParameters
Parameters which will be passed to the startup program

Figure 6.17: *Configuration of the "Automatic Installer"*

Figure 6.17 shows the configuration settings for the AVL IndiCom Version 2.3 installation. The tool periodically scans the defined remote path for a new build folder. Since the names of the build folder can change, the search string is expressed as a regular expression. Before the installation is executed, the license files have to be copied and some configuration values have to be set, which is illustrated in the "Setup Settings" section. After the installation, various post-installation tasks as setting conguration values have to be done and the new application is started. All properties are well documented and a description of the property is shown below when the item is selected. The settings can be saved as XML file and transferred to another testing environment.


IndiCom automatic installer

IndiCom1.6 AutoTest 671 | IndiCom2.3 AutoTest 671

	Build Version	RemotePath	Date	Status	
▶	Build 324	\\Atgrzwd508640\I...	03.03.2014 13:14:12	10%	Cancel
	Build 225a_FINAL	\\Atgrzwd508640\I...	12.11.2013 12:13:01	0%	Install
	Build 293_Daimler	\\Atgrzwd508640\I...	12.11.2013 12:09:01	0%	Install
	Build 323	\\Atgrzwd508640\I...	12.11.2013 12:03:01	0%	Install
	Build 116_Final	\\Atgrzwd508640\I...	12.11.2013 11:54:32	0%	Install
	Build 315_HF7	\\Atgrzwd508640\I...	12.11.2013 11:45:00	0%	Install
	Build 239a_HF1	\\Atgrzwd508640\I...	12.11.2013 11:31:00	0%	Install
	Build 290a_HF5	\\Atgrzwd508640\I...	12.11.2013 10:59:29	0%	Install
	Build 322_HF8	\\Atgrzwd508640\I...	12.11.2013 10:37:28	0%	Install
	Build 284a_HF4	\\Atgrzwd508640\I...	12.11.2013 10:30:58	0%	Install
	Build 260_4CA1_...	\\Atgrzwd508640\I...	12.11.2013 10:24:57	0%	Install
	Build 213a_Suite	\\Atgrzwd508640\I...	12.11.2013 10:06:27	0%	Install
	Build 299_HF6	\\Atgrzwd508640\I...	12.11.2013 10:03:57	0%	Install
	Build 269a_HF3	\\Atgrzwd508640\I...	12.11.2013 09:58:27	0%	Install
	Build 254a_HF2	\\Atgrzwd508640\I...	12.11.2013 09:47:56	0%	Install

Copying remote files to setup directory ...

Figure 6.18: *Automatic installation routine*

Figure 6.18 shows the automatic installation of the last available AVL IndiCom build. Only the latest available version is installed, older versions are ignored. The program contains full logging in order to detect installation problems. The tool will be started with the operating system and will then run as a background task in the windows task bar, so that the current installation status can be checked at any time.

6.6.2 Compatibility to different measurement devices

The main problems when porting the automatic test to a new hardware, were incompatible references or the determination of the correct tolerance values for the comparison. With the implementation of a new signal comparison method it was possible to solve these problems. In the current state, the automatic test is prepared for an arbitrary number of new hardware devices.

Chapter 7

Field Test

At the time of publication of this thesis, the data migration of the test cases to the 'HP Quality Center' had not been finished. A small set of test cases as well as some equivalence classes have already been created.

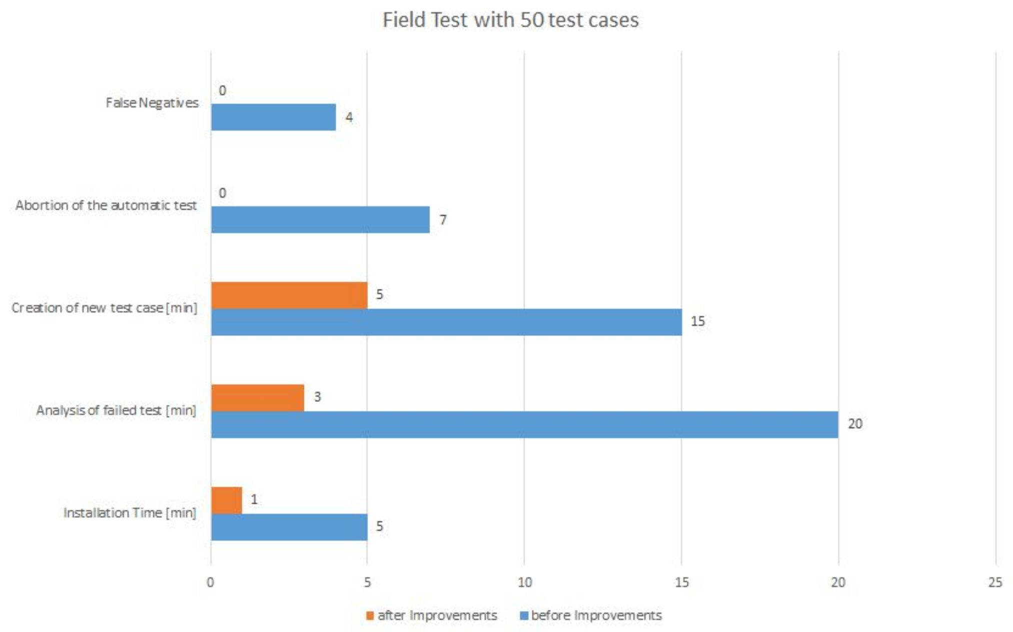


Figure 7.1: *Comparison of the automatic test and improvements*

In figure 7.1 a comparison of some common tasks related to the automatic test is shown. The evaluation shows promising results. For the same set of 50 test cases the detection of 'false negatives' shows, that with the improved comparison method it was possible to eliminate this problem. Errors in the implementation of the automatic test and program crashes of AVL IndiCom lead to the abortion of the automatic test. By outsourcing the test control, the automatic test now runs stable and no cancellation of the test routine could be detected.

The creation of new test cases was an expensive task, since new scripts had to be created and each reference had to be generated manually. A comparison shows that the time for creating a new test case could be reduced to one third of the time required before. Furthermore, the analysis of failed tests could be considerably reduced, because there is no need of searching through log files any more and no debugging of source code has to be done. The initialisation of the automatic too, was simplified. The manual installation routine took about 5 minutes and this had to be done on each test environment for each measurement device. Now, the installation is executed automatically and simply needs to be checked.

In this early stage of launching the improvements, new errors were already detected, which resulted simply from changing the test case order of the test cases. The execution of measurements with a high-resolution scope produces huge data sets. If such test cases are executed one after another the result are memory problems of AVL IndiCom. The recording of the performance figures with “System Observer” tool facilitated the analysis of the problems and some of them could already be solved. The commissioning of a new test environment for a new measurement hardware was also realized without any problems.

The conversion of the test cases into the XML format currently is still in progress and will continue to take time and occupy test resources. However, this is a unique task and the advantages are obvious, since each test case now has to be documented and the administration can be done in ’HP Quality Center’.

Chapter 8

Conclusion

First this master thesis was intended to analyse the benefit of the automated test system and to verify the test methods. Some studies (see chapter 4) showed, that the automated test is too sensitive to environmental changes and is unsuitable for different measurement modes and different measurement devices. An analysis of the runtime and stability put the benefit of the automatic test into doubt. Therefore, an investigation on alternative evaluation methods and improvements of the test process was accomplished.

The outcome showed, that an improved comparison of the measurement results reduced the amount of software errors that were detected by mistake. With the implementation of a new comparison method the stability of the automatic test could be improved and new errors were detected. With the use of new tools and the proprietary development of supporting tools it was possible to improve the performance of the automated test system. A connection of the automatic test to the "HP Quality Center" facilitates the evaluation and administration of the test cases and results and meets the requirements for a standardized and centralized test management within the entire company.

The provided tools described in this master thesis facilitate working with the automated test and raise awareness of the team members regarding the importance of test automation. The automatic test is now a set component of the development process.

An automatic test is certainly not a sufficient method to improve the quality of a software product. Tests during development or test driven development mechanisms should be promoted, since they are State-of-the-Art. During the software development process, multiple steps of testing can develop, starting with component tests, integration tests and finally a system test or, if required, an acceptance test. All these steps can contain redundant checks for the same feature in the software. For this reason, the coordination and planning of the test routine is a key factor for the efficiency of the test and the quality of the software.

Almost all steps mentioned above can be automated, from component tests with xUnit[1], to system tests with “Capture and Replay” tools like UIA[2].

Advantages and disadvantages of the manual test:

+ Usability testing
+ Intuitive testing
+ Detailed testing
- High personnel costs
- High costs over the whole development period
- Formal test instructions are needed

Advantages and disadvantages of the automatic test:

+ Test results are reproducible
+ Faster than human
+ Always available
- High initial (development) costs
- High costs over the whole development period

With respect to the advantages and disadvantages of the manual and the automatic test, it can be assumed that it is worth to invest in a test automation system, when considering the financial and quality aspects. Assessing the whole development time for a requirement, the test cases can be executed and repeated on a regular basis and in a more efficient way. The development can be done during office hours and the testing is executed over night. Of course, the development of an automated test system requires additional capital costs, but on a long term view it can reduce the overall development costs.

In summary, it would appear that everything that can be tested automatically, should be tested automatically. This however bears the risk of loosing sight of tests that cannot be automated, like the usability of the software or the intuitive testing. An important advantage of the manual test is the direct intervention as well as the execution of intuitive tests and the benefit of the expert knowledge of the end

[1] xUnit is the collective name for several unit testing frameworks
[2] Microsoft UI Automation, Application Programming Interface

user. A manual test can test “beyond the obvious” and detect vulnerabilities which influence the quality of a software to a greater higher detail.

But, a test automation must not be economically viable. If, in the long-term view, no money can be saved, the automation is an end in itself. For example, the execution of a test takes 10 minutes in average. The test case can be automated within one month of development and then only takes just under 5 minutes.

$$\begin{aligned} 10x &= 9600 + 5x \\ x &= 1920 \end{aligned} \tag{8.1}$$

Equation 8.1 shows the calculation for the example with an average working time of 160 hours (9600 minutes) per month.The result for x shows that 1920 runs are required for the test automation to become profitable. If the test runs once per day, it will pay off only after 5 years (see figure 8.1).

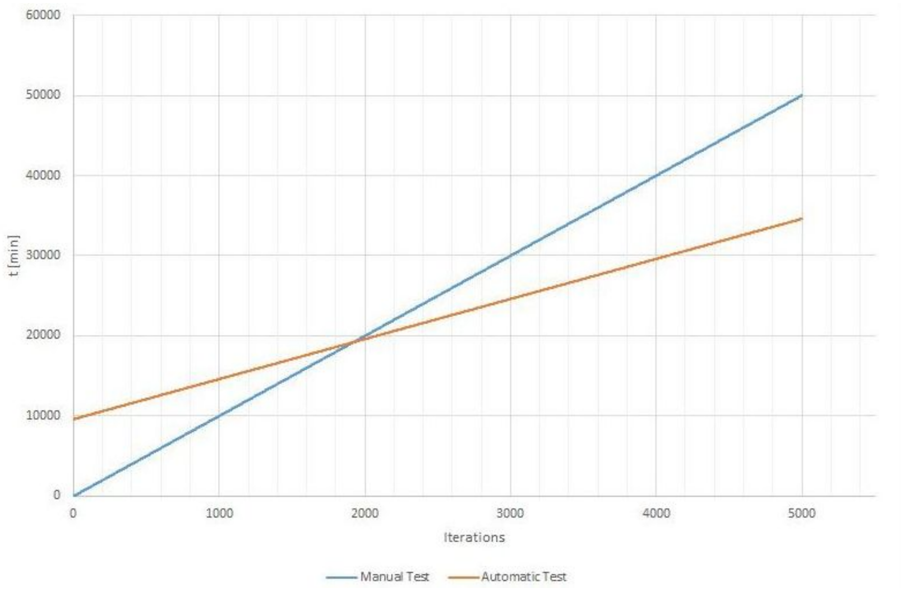


Figure 8.1: *Pay-off of an automatic test case after 1920 iterations*

If we assume that the same test only takes 1 minute when automated, then the threshold will decrease to 1067 iterations as shown in equation 8.2.

$$\begin{aligned} 10x &= 9600 + 1x \\ x &= 1066,667 \end{aligned} \tag{8.2}$$

This result is better but still not satisfying, since the test case will pay-off after nearly 3 years (see figure 8.2).

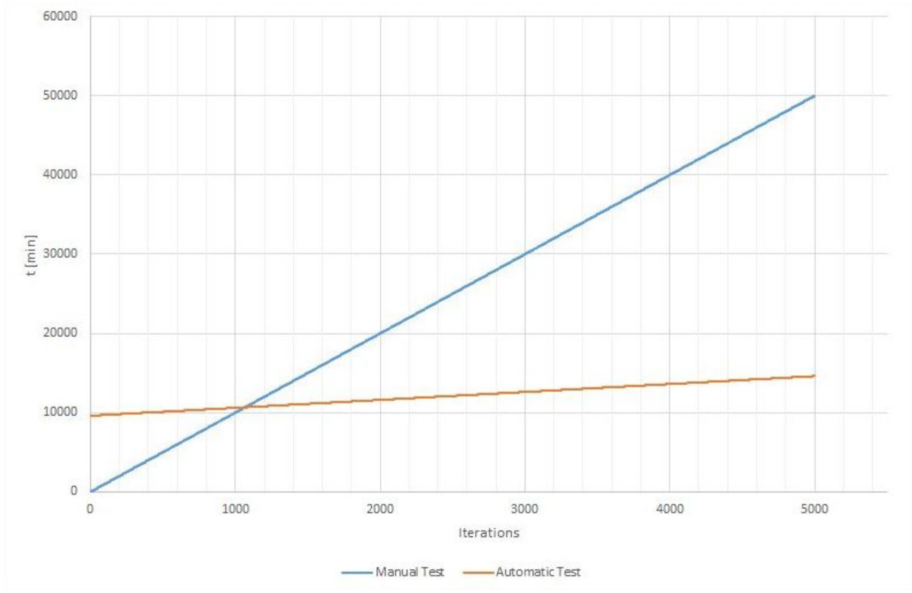


Figure 8.2: *Pay-off of an automatic test case after 1067 iterations*

If the development time for the automation of this test is reduced to 1 week, then the automation will pay off after 267 iterations as shown in equation 8.3.

$$\begin{aligned} 10x &= 2400 + 1x \\ x &= 266,667 \end{aligned} \tag{8.3}$$

Figure 8.3 shows that for a short development time, the amortization period could be reduced to less than one year.

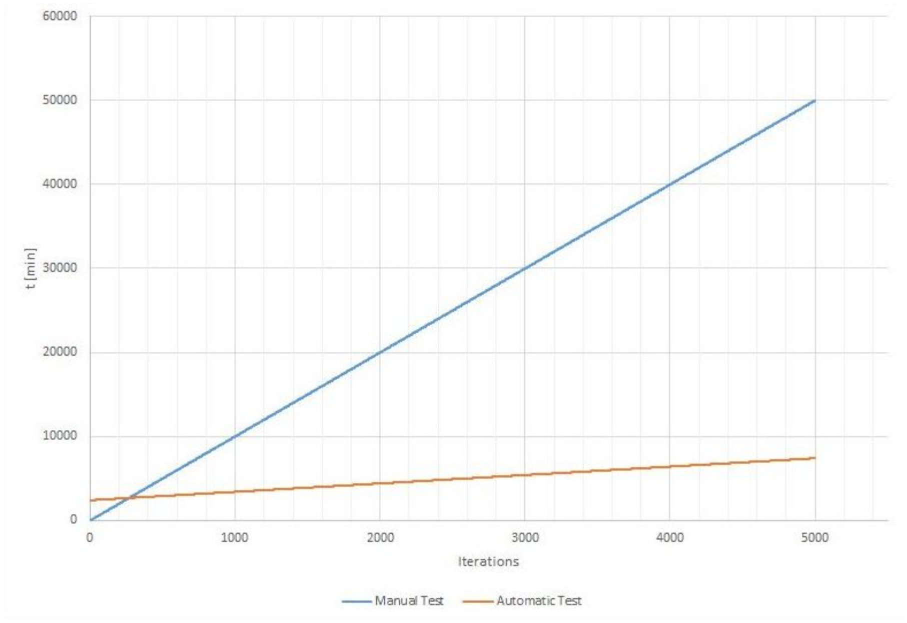


Figure 8.3: *Pay-off of an automatic test case after 267 iterations*

Therefore, the automation will be a good investment if the test is executed every day for a long term. The examples above show, that the effort for creating an automated test and the execution time influence the economic efficiency of the automation. If tests are difficult to automate or the software does not provide automation mechanisms, the meaningfulness of the automation has to be put into question.

Chapter 9

Outlook

The field test of the improved automatic test showed that it was possible to detect new errors and the stability of the test environment was improved. For the future usability and reliability of the automatic test system, a consequent maintenance of the test cases and test plans is necessary. Through the simplification of the test creation process, new test cases will be generated but nevertheless, the automation of every test requirement still will not be possible. Small changes in the software, which do not have a high potential of causing an error, do not need to be tested manually over and over again while complex and variable procedures can be automated.

The introduction of new agile software development methodologies requires an increased degree of automation. The efforts for standardization of the tools used for software testing in the company will require the implementation of new extensions and interfaces to other test tools.

A topic which was renounced in this master thesis is the automated GUI[1] testing. The introduction of an automation is considered however, since not all measured results can be displayed, the verification of the measured data is indispensable.

[1] Graphical User Interface

Appendix A

Code Listing

A.1 Tolerance Area [MATLAB]

```matlab
function [upperBand, lowerBand] = ToleranceRange(signal, scale)
% Calculates the tolerance range for a signal
% signal      The reference signal
% scale       Scale factor for \mu (per thousend)
% returns:
% upperBand   upper tolerance border for this signal
% lowerBand   lower tolerance border for this signal

    refMax = max(signal);
    refMin = min(signal);

    range = refMax - refMin;
    tolerance = (range / 1000) * scale;

    upperBand = zeros(size(signal));
    lowerBand = zeros(size(signal));

    upperBand = upperBand + (refMax + tolerance);
    lowerBand = lowerBand + (refMin - tolerance);

end
```

A.2 Pearson Correlation Coefficient [Concerto]

```
// Pearson Correlation Coefficient
// value btw. -1 and 1 showing the similarity of 2 datasets.
// Arguments:
//                X - original dataset
//                Y - distorted dataset
//        Threshold -
// Return:
//                -1 the two datasets are inverse
//         0 the two datasets are totally different
//         1 the two datasets are identical
arg X, Y, From=1, To=0
        TINY = 0.000000000000001 //1.0e-15 to prevent a division
            trough zero
        sumXX = 0
        sumYY = 0
        sumXY = 0
        if (To = 0) then
                To = X.Count
        endif
        if(From > X.Count or From > Y.Count) then
                // Invalid Startpoint
                TraceError("CORR: Invalid Startpoint, FROM:=" +
                           From + ", X:=" + CStr(X.Count) + " Y
                              :=" + CStr(Y.Count))
                return 0
        endif
        if(To > X.Count or To > Y.Count) then
                // Invalid Endpoint
                TraceError("CORR: Invalid Endpoint, TO:=" +
                           To + ", X:=" + CStr(X.Count) + " Y:="
                               + CStr(Y.Count))
                return 0
        endif
    avgX = Avg(X)
    avgY = Avg(Y)
        for i = From to To
                tX = X.y[i] - avgX
                tY = Y.y[i] - avgY
                sumXX = sumXX + (tX * tX)
                sumYY = sumYY + (tY * tY)
                sumXY = sumXY + (tX * tY)
        next i
        div = sqrt(sumXX*sumYY) + TINY
        r = sumXY / div
return r
```

A.3 White Gaussian Noise Analysis [MATLAB]

```matlab
function [noiseAnalysis] =
NoiseAnalysis(signal, minSigma, maxSigma, stepSize, sampleCnt)
% Generates samples with White Gaussian Noise and
% compares the results with statistical measures
% signal      The reference signal
% minSigma    The minimum /sigma ot the range
% maxSigma    Upper bound of the /sigma range
% stepSize    The step size for /sigma
% sampleCnt   Amount of samples which will be
%             generated
% noiseAnalysis  A result matrix with the statistical values
%                as columns and one row forech /sigma
    range = minSigma:stepSize:maxSigma;
    range = transpose(range);
    overall = zeros(size(range));
    min = zeros(size(range));
    max = zeros(size(range));
    mean = zeros(size(range));
    std = zeros(size(range));
    correlation = zeros(size(range));
    resultMatrix = [range, overall, min, max, mean, std,
       correlation];
    for set = 1:1:sampleCnt
       for i = 1:1:length(resultMatrix)
           sigma = resultMatrix(i,1);
           noise = sigma * randn(size(signal));
           compare = signal + noise;
           %Calculation of statistical values
           [result, value, failMin, failMax,
            failMean, failStd, corr, text] = evaluate(signal,
               compare);
           if result == 0
              resultMatrix(i,2) = resultMatrix(i,2) + 1;
           end
           if failMin == 1
              resultMatrix(i,3) = resultMatrix(i,3) + 1;
           end
           if failMax == 1
               resultMatrix(i,4) = resultMatrix(i,4) + 1;
           end
           if failMean == 1
               resultMatrix(i,5) = resultMatrix(i,5) + 1;
           end
           if failStd == 1
               resultMatrix(i,6) = resultMatrix(i,6) + 1;
           end
           resultMatrix(i,7) = resultMatrix(i,7) + corr;
       end
    end
    for i = 1:1:length(resultMatrix)
        resultMatrix(i,7) = resultMatrix(i,7) / sampleCnt;
    end
    noiseAnalysis = resultMatrix;
end
```

Bibliography

[AD97] Larry Apfelbaum and John Doyle. Model based testing. In *Software Quality Week Conference*, pages 296–300, 1997.

[AKSS03] Alain Abran, Adel Khelifi, Witold Suryn, and Ahmed Seffah. Usability meanings and interpretations in iso standards. *Software Quality Journal*, 11(4):325–338, 2003.

[AMKS08] G.R. Askari, N. Motamedi, M. Karimian, and H. Sadeghi. Design and implementation of an x-band white gaussian noise generator. In *Electrical and Computer Engineering, 2008. CCECE 2008. Canadian Conference on*, pages 001103–001106, May 2008.

[BAJ93] G. Bazzana, O. Andersen, and T. Jokela. Iso 9126 and iso 9000: friends or foes? In *Software Engineering Standards Symposium, 1993. Proceedings., 1993*, pages 79–88, Aug 1993.

[Bei95] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems.* Wiley, 1995.

[Bur03] I. Burnstein. *Practical Software Testing: A Process-Oriented Approach.* Springer Professional Computing. Springer, 2003.

[FHRF12] Syahrul Fahmy, Nurul Haslinda, Wan Roslina, and Ziti Fariha. Evaluating the quality of software in e-book using the iso 9126 model. *International Journal of Control & Automation*, 5(2), 2012.

[Gmb11a] AVL List GmbH. *AVL IndiCom 2011 - Formula/Script Editor*, 2011.

[Gmb11b] AVL List GmbH. *AVL IndiCom 2012 - Exploration Guide*, 2011.

[Gmb11c] AVL List GmbH. *IndiCom Mobile 2012 - Users's Guide*, 2011.

[HEK05] J. Hartung, B. Elpelt, and K.H. Klösener. *Statistik: Lehr- und Handbuch der angewandten Statistik ; mit zahlreichen, vollständig durchgerechneten Beispielen.* Oldenbourg, 2005.

[LBN08] F.M. Lord, A. Birnbaum, and M.R. Novick. *Statistical Theories of Mental Test Scores*. Addison-Wesley series in behavioral science: quantitative methods. Information Age Pub, 2008.

[Lig09] Peter Liggesmeyer. *Software-Qualität, Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, 2009.

[LRN88] Joseph Lee Rodgers and W Alan Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42(1):59–66, 1988.

[Mic] Microsoft. MSDN microsoft developer network. `https://msdn.microsoft.com/de-de/library/system.diagnostics%28v=vs.80%29.aspx`. Accessed: 2015-01-30.

[OB10] M.S. Obaidat and N.A. Boudriga. *Fundamentals of Performance Evaluation of Computer and Telecommunications Systems*. Wiley, 2010.

[O'R14] Gerard O'Regan. *Introduction to Software Quality*. Springer International Publishing, 2014.

[Pha00] H. Pham. *Software Reliability*. Springer, 2000.

[Pis02] Rudolf Pischinger. *Engine indicating, User Handbook*. AVL List GmbH, 2002.

[Pre07] Anton Prettenhofer. Automatisierter Test eines Programms zur Messdatenverarbeitung. Master's thesis, Campus02 Graz, 2007.