

OCTane - Applying software quality-measures to bare-metal firmware

Florian Hinterleitner



MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

embedded systems design

in Hagenberg

im Juni 2022

Advisor:
Langer, Rankl, Zorin

© Copyright 2022 Florian Hinterleitner

This work is published under the conditions of the Creative Commons License *Attribution-NonCommercial-NoDerivatives 4.0 International* (CC BY-NC-ND 4.0)—see <https://creativecommons.org/licenses/by-nc-nd/4.0/>.

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere. This printed copy is identical to the submitted electronic version.

Hagenberg, June 15, 2022

Florian Hinterleitner

Contents

Declaration	iv
Abstract	vi
Kurzfassung	vii
1 Introduction	1
1.1 Motivation	1
1.2 Optical coherence tomography	1
1.3 Galvanometer-Scanners	2
1.4 Control of Galvanometer-Scanners	3
2 Fundamentals - Code Quality and Real-time	6
2.1 Motivation	6
2.2 Terminology and definitions of terms	8
2.2.1 Quality, Quality requirements, Quality features, Quality measures	8
2.2.2 Error, Failure, Fault	9
2.2.3 Correctness	9
2.2.4 Completeness	9
2.2.5 Testability	9
2.2.6 Safety and Security	9
2.2.7 Reliability	10
2.2.8 Maintainability	10
2.2.9 Availability	10
2.2.10 Robustness	10
2.2.11 Dependability	10
2.3 Function-oriented Testing	11
2.3.1 Equivalence class partitioning	11
2.3.2 Boundary value analysis	13
2.3.3 State based testing	13
2.4 Coverage metrics	13
2.4.1 Properties and goals	14
References	16
Literature	16

Abstract

The company RECENDT GmbH develops and produces OCT-systems (optical coherence tomography), for which, as part of this master thesis, a part of the controlling system will be designed. For 2-dimensional measurement with OCT-systems, galvanometer-scanners are employed in an x/y-mode. These are highly-dynamical rotational drives for optical application with approximately 20° of angle for forward and back-rotation at rates up to 1kHz. Carrying mirrors, that are rotating along, these galvanometer-scanners can manipulate an optical path, typically from a focused coherent light-source, in 2 dimensions and therefore allow to scan whole surfaces with OCT-systems.

The chosen scanner-models require control signals to create scan-patterns, usually rectangles. Therefore, two synchronous ramp signals, a slow and a fast one are necessary. The task at hand is now, to program an existing microcontroller-hardware to form a two-channel arbitrary signal generator, called OCTane. This contains firmware modules to access digital-analogue-converters, trigger-units, for correct timing and the synchronisation. Furthermore USB-connectivity in a SCPI-style fashion has to be implemented, as the control of the unit shall be provided via USB.

Well established measures of quality for software development allow insightful analysis of the inspected software. These measures, for the bigger part, rely on an underlying operating system, and the inspected software to be compiled on the host-system or an equivalent one. To gain similar insight to cross-compiled bare-metal firmware, the application of mentioned quality-measures shall be researched. Obstacles are to be expected from the necessary cross-compilation and the absence of an underlying operating-system.

Kurzfassung

Die Firma RECENDT GmbH entwickelt und baut OCT-Systeme (optical coherence tomography), für die im Rahmen dieser Masterarbeit ein Teil der Steuerung entworfen werden soll. Zur 2-dimensionalen Messung mit OCT-Systemen kommen Galvanometer-Scanner im X/Y-Betrieb zum Einsatz. Das sind hochdynamische Drehantriebe für optische Anwendungen, die mit einer Rate von rund 1kHz etwa 20° vor- und rückwärts rotieren können. Sie tragen mitrotierende Spiegel um den optischen Pfad in 2 Dimensionen auszulenken und somit flächige Scans zu ermöglichen.

Die ausgewählten Galvanometer-Scanner benötigen Steuersignale zur Erzeugung der Scan-Muster. Typischerweise sind dies zwei synchrone Rampen-Signale, eines schnell, eines langsam. Aufgabe ist es nun, auf bestehender Mikrocontroller-Hardware einen 2-kanaligen arbiträren Signalgenerator, namens OCTane zu programmieren. Dieser soll sowohl Rampen-Signale als auch arbiträr gewählte Signalformen erzeugen können. Dies beinhaltet FW-Module für die Digital-Analog-Wandler, Trigger-Einheit für das Timing sowie Synchronisation der Kanäle. Weiters ist die USB-Kommunikation per SCPI-Protokoll zu programmieren. Die Anbindung an eine übergeordnete Steuerung des OCT-Systems erfolgt per USB.

Etablierte Qualitätsmaße der Software-Entwicklung erlauben tiefe Einblicke in die untersuchte Software. Diese Maße stützen sich, größtenteils, auf ein unterliegendes Betriebssystem, sowie die Tatsache, daß die untersuchte Software auf dem Zielsystem oder einem Äquivalenten kompiliert wird. Um ähnlich tiefgreifenden Aufschluss über cross-kompilierte bare-metal Firmware zu erhalten, soll die Anwendbarkeit erwähnter Qualitätsmaße untersucht werden. Als hinderlich dabei ist zu erwarten, daß Firmware generell cross-kompiliert werden muss und eventuell kein unterstützendes Betriebssystem enthält.

Chapter 1

Introduction

1.1 Motivation

The RECENDT GmbH researches, develops and produces, among other technology areas, measurement systems employing optical coherence tomography. A key element of such OCT-systems are galvanometer-scanners. These allow for investigation of areas, instead of only point-wise measurements, by manipulating a laser-beam. This manipulation again, has to be controlled via two separate steering-voltages, one for manipulation in x- and y-axis. An existing microcontroller-board, providing sufficient precise and fast DACs, is to be programmed, to form the 'OCTane', a signal-generator for mentioned steering-voltages, controllable via USB. The resulting Firmware shall also incorporate a HAL, utilizing several other functionalities, the microcontroller has to offer. Optionally, adapted curves for the steering voltages shall be investigated to allow linear control over galvanometer-scanners in higher frequency ranges.

1.2 Optical coherence tomography

Optical coherence tomography (OCT) is an imaging measurement method for the analysis of transparent and semi-opaque materials and shows similarities to the measurement processes via ultrasound or radar. The sample to be measured is subjected to an electromagnetic wave, the resulting 'echoes' are analysed with regard to their times of flight. From these run-times again, the geometric structure is determined, including the layer structure of the sample and also the maximum penetration depth of the applied wave. This creates measured point with an in-depth resolution. Usually this EM wave is of a coherent broadband light source in the visible up to the near infrared spectrum. Coherent means, that several wave-bundles of a light source must have a fixed phase relationship to each other. This is necessary to obtain stable interference patterns. For the Detection of the echoes, however, conventional photodetectors or cameras do not suffice, on the one hand due to the propagation speed of light, on the other hand due to the low reflected light intensities. Therefore Interferometry is used to determine the depth of penetration of a photon, being reflected from the sample. In interferometry, a laser beam is split into two waves of half the initial optical power. One wave is sent on an optical reference path of known length, the other to the surface of the sample. The

reflections, the returning waves are superimposed and, depending on the nature of the sample material, result in constructive or destructive Interference. This interference can be detected using a photodetector[1], or a spectrometer[2] and used for further processing. A single measured point and its depth information about the material under test is called an A-scan. Aggregates a lot of A-scans along a line (X-direction) across the sample material, forms a B-scan and the aggregation of B-scans along a line in the Y direction a volume scan, i.e. a spatial, three-dimensional image of the sample material. Relevant parameters of OCT systems are the penetration depth, the axial and lateral measurement range, axial and lateral resolution and the measurement speed. While the penetration depth of ultrasound typically reaches a few centimetres and a resolution in the millimetre range, OCT allows only to look a few millimetres below the surface, but with micron resolutions. Measurable areas, or field-of-view, in ultrasound is in the order of centimetres, with OCT in the order of millimetres[1]. achievable speed al results from A-scan rates up to 100kHz. The term 'optical coherence tomography' results on the one hand from the coherent light source. The other two parts of the name, 'tomos' means slice or section, and 'graphein' stand for writing or drawing, and both come from Greek. They reflect that the resulting image is assembled from individual slices or sectional images. The manipulation of the light beam along the mentioned lines takes place with rotatably mounted mirrors, one for the X one for the Y direction. The faster this rotation is possible, the faster OCT-images can be created. One widespread technical realization, allowing very fast rotation of the mirrors called a galvanometer-mirror or -scanner.

1.3 Galvanometer-Scanners

Galvanometer scanners (colloquial: galvos) are highly dynamic opto-mechanical components, based on the classic galvanometer according to Hans Christian Oersted: A rotatable, magnetizable object, e.g. a magnetic needle, that will be deflected from its position in the proximity of a current-carrying conductor. The low sensitivity of the effect on the current is improved by a high number of windings of the electric conductor around the deflectable object, creating an electrical Inductance, a coil. The non-linear connection between current and deflection angle can be linearized to a first-order approximation, by placing the coil between a rigidly positioned iron-cylinder inside and a permanent-magnet, which is arranged outside the coil[2]. If this classic Galvanometer is equipped with a mirror as a rotatable object, optical paths, specifically: the beam-path of a point light source, can be manipulated in one space dimension. Feasible for technical applications is, that this manipulation can be controlled reasonably linear by the current or voltage at the galvo coil. With the galvanometer-scanner employed for this master-thesis, power-electronics for the conversion of control signals to the required coil-currents and -voltages were already included. Therefore, furthermore, only 'control signals' will be discussed, instead of currents and voltages. A combination of two galvanometer scanners in a suitable geometric arrangement, irradiated with a point laser, allows to manipulate this point in two dimensions. Such an arrangement is shown in fig. []. At sufficient speed with which the laser point is deflected, 2-dimensional contours can be projected, resulting in a 'stationary' image for the human eye. It shall be noted, that only closed geometric figures are possible, as long as the light source itself cannot

be turned off. These components are commercially available as laser scanner and used for light effects at music events, art installations and in discotheques. Applied to OCT-systems, on the other hand, galvanometer scanners are used to expand measurement from a single point of interest. Deflecting mentioned coherent light source of an area of an examined sample, allows for two-dimensional analysis of the sample. If a dedicated x- and a y-galvo are to be steered with a slow and a fast ramp, respectively, this results in a rectangle on the sample, which is the desired 'image shape' of the laser dot for OCT systems. In this way, samples can be scanned in a grid pattern.

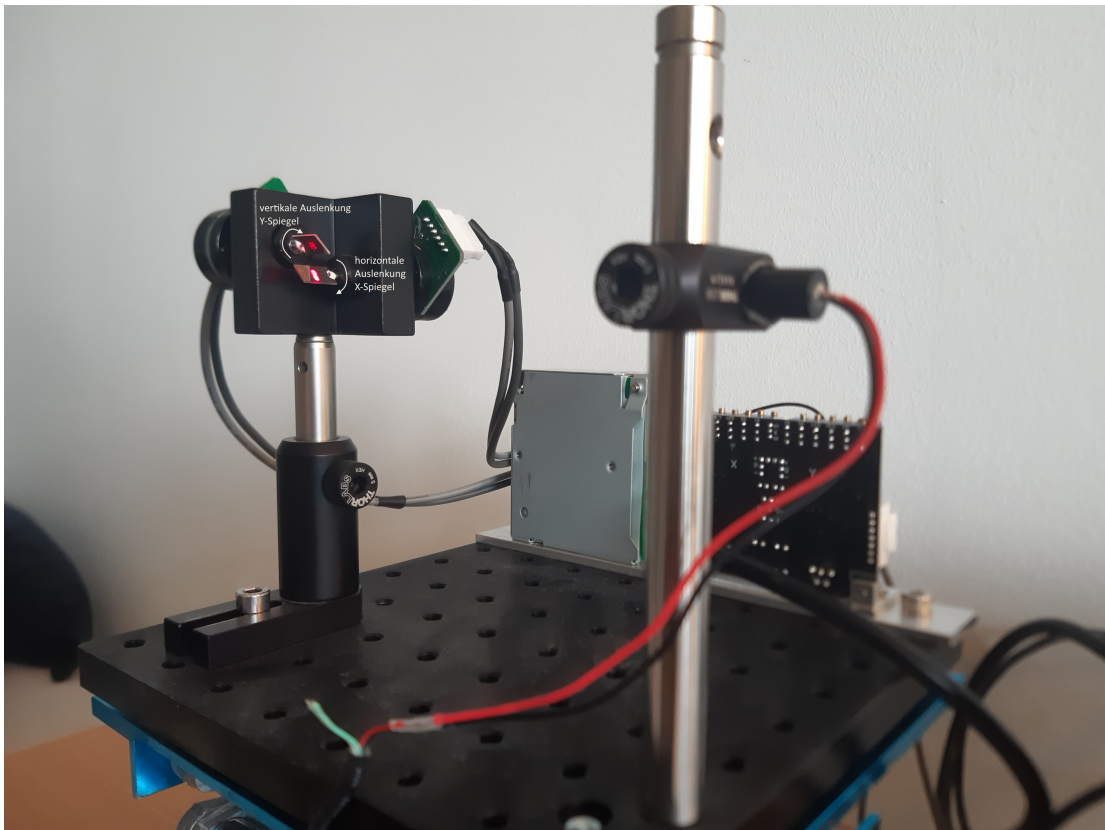


Figure 1.1: DetailGalvoOn

1.4 Control of Galvanometer-Scanners

Commercially available Galvanometer-Scanners usually allow control in the form of analogue voltage inputs with a range of ± 10 Volts. The angle of the rotated mirror follows that control-voltage in a linear manner for sufficiently low frequencies. The signal-forms to result in the rectangular scan-grids, as described in the previous section, are depicted in Fig. 1.3. To achieve these signals, a microcontroller-board was designed, including 16-bit-DACs, USB-PHY and coaxial Trigger-IOs, among other features. To utilise these features and form an arbitrary signal generator for mentioned ramp-signals, a firmware is necessary. This should be done in a fashion employing quality-assurance

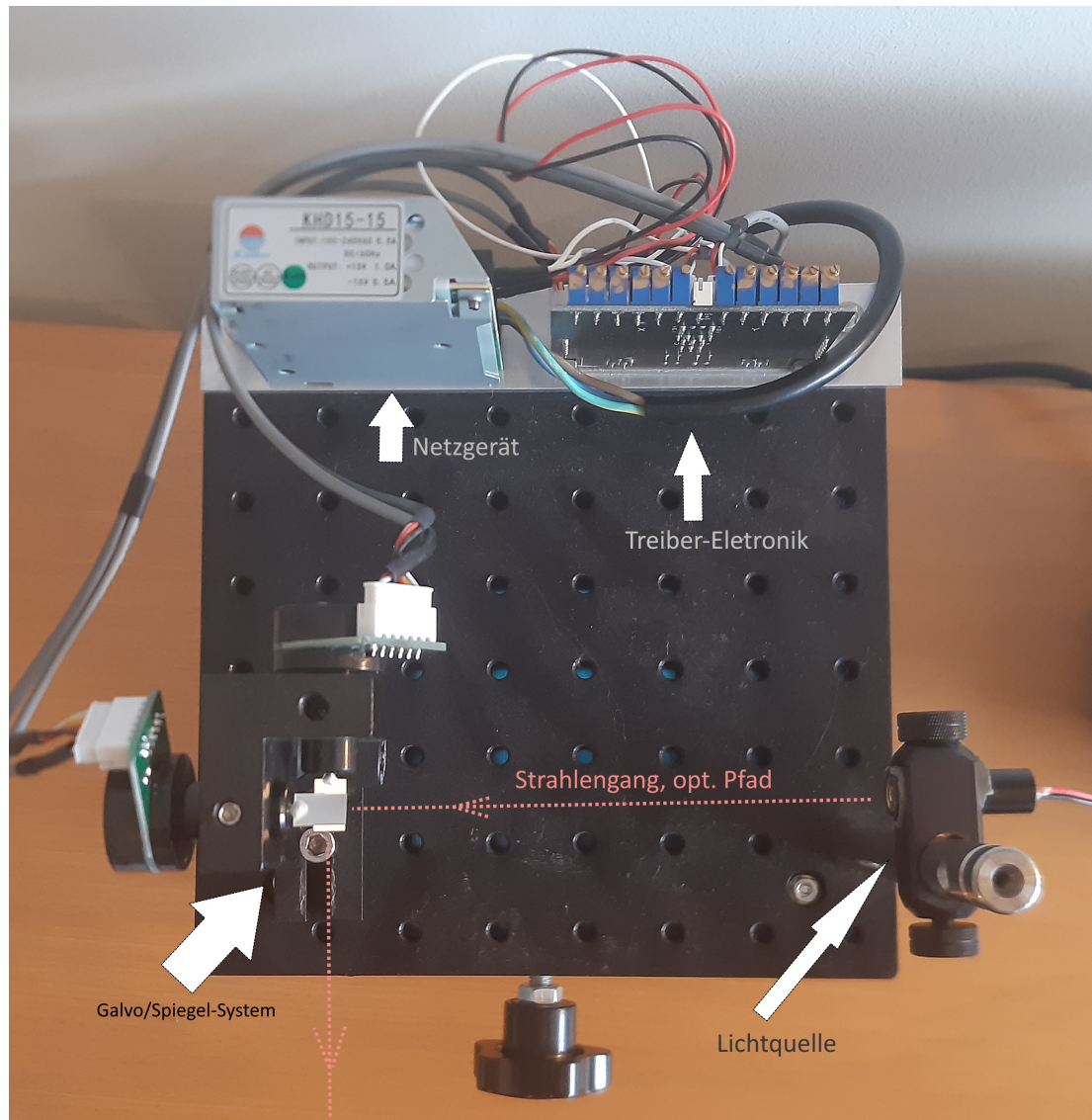


Figure 1.2: DutTop02

during the implementation-process, as well as the verification-phase of the firmware and its supporting hardware. Aside from signal-generation and USB-connectivity, additional features are desirable, such as user-controllable Relays, utilisation of watchdog-timer, UART-, I2C- and SPI-ports as well as analogue inputs. The combination of hard- and firmware will be called OCTane. **To-Do:** block-schematic: PC -> OCTane -> 2xDAC -> Galvo-Driver

This leads to the scientific problem at hand:

How can measures of quality be applied to a bare-metal firmware?

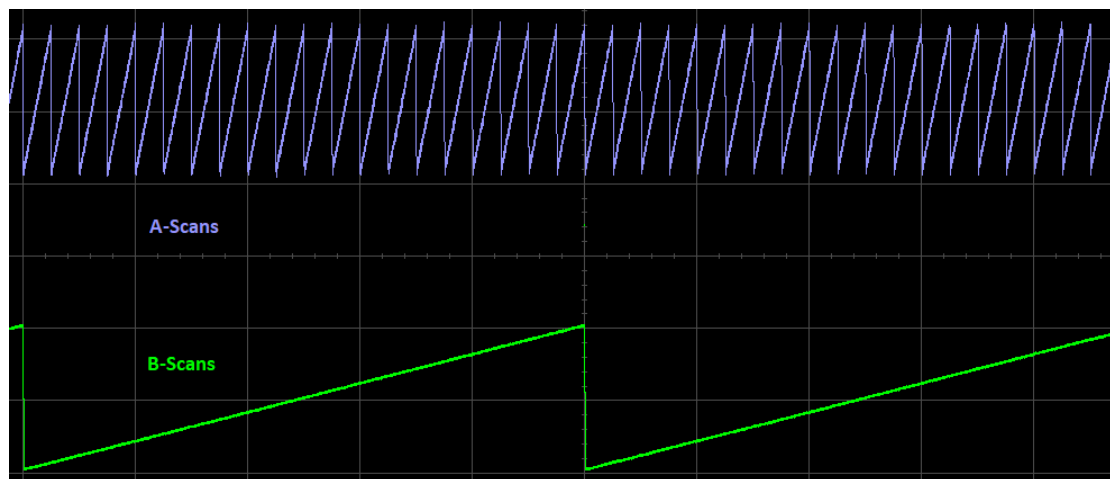


Figure 1.3: GalvoRamps01

Chapter 2

Fundamentals - Code Quality and Real-time

2.1 Motivation

As the areas of applications for microcontrollers, as well as the complexity of their software continues to grow, also the demands towards correct and reliable function of those systems, increase accordingly. This causes the efforts, put into software-development, to usually surpass efforts for the corresponding hardware. As the software's lifespan also surpasses that of the hardware, these efforts becomes a sensible investment. On the other hand, to obtain the most value from that software-product, it has to fulfil certain quality measures. Neglecting these quality aspects would lead to an unjustifiable amount of work necessary for maintenance, when the product is already in service. Investigating the dynamics of software development efforts leads to the result, that, the later in a project lifecycle, changes become necessary, the higher the expenses. More detailed expressed, errors are induced in the implementation, persist unnoticed through testing and verification. Finally, this errors appear as faults in operation at the customer site. Furthermore, if the project suffers from lazy documentation, insufficient structure, identifying these errors and correcting them becomes consuming in time, cost and resources. This phenomenon is further escalated with the increasing complexity of nowadays software. An even worse phenomenon can arise: Insufficient understanding of a faulty piece of software at hand, can lead to introducing even more errors with additional code, that is actually intended to fix a bug. Especially when poor structuring masks hidden dependencies between modules. A general rule of thumb is: The earlier an error originates, for example in the phase of gathering requirements, the more extensive changes are necessary, later on. In other words: The earlier an error is induced and the later it is discovered, the more expensive are its consequences. Achieving a sufficient quality level from the beginning of the project, can significantly reduce a waste of money, hours and enthusiasm on unnecessary maintenance and also substantially reduce the customers disapproval.

But these problems have suitable solutions at hand. It is not compulsive to plunge money, hours and employee motivation on unnecessary maintenance. Employing the right methods, implemented software can become reliable, easy to change, inexpensive to

maintain and allow a more intuitive understanding. Distinguishable into two categories, these methods are either analytical or constructive. Best practice is to employ a circular combination of constructive and analytical methods. Constructive means alone can assist in preventing errors in the intended product, but not guarantee their absence. Analytical means are capable of demonstrating the absence of errors, but not their prevention, so a large amount of preventable errors might emerge, when only analytical means are put to use. The combined use advisable would be, to employ constructive methods during every phase of a development project, and assessing the intermediate results with analytical methods by the end of each phase. This process is called the 'principle of integrated quality assurance' [0]. If these intermediate results do not meet the arranged quality criteria, the current state of the project is not passed on to the next phase, but the current phase has to be extended. This implies, that the current state of the product requires further development, until all necessary criteria are met. This phase- and quality driven conduct, supports the development team in detection of errors at an early point and their removal at reasonable effort. Ideally all errors induced in a development phase are detected and eliminated by the end of the same phase. This should further help in minimizing the number of errors in a product over several phases.

The described process makes it evident, that testing only a finished product is no sufficient way of ensuring high quality. Already the first intermediate result has to be investigated for deviations from the quality goals and measures have to be taken for correction at an early stage. Also an integration of constructive and analytical quality measures is required. While constructive methods are advised during the implementation activities of a phase, it should be followed by the corresponding analysis.

A key factor in ensuring the intended quality lies in the early definition of these quality goals. It constitutes not of defining the requirements, but the specification of the desired quality features. This has to happen even before the phase of requirement-definition, as the requirements themselves are affected by aforementioned quality goals. On the other hand, testing results against quality features is also of central importance. The typical approach of every developer is, to call a written program with a few sets of inputs and observe the program for expected, or divergent behaviour. This already constitutes for an informal dynamic test. Inspecting the code after, implementing it, for structural errors is the informal equivalent of a static analysis. As these informal methods are widespread among programmers, employing formal processes of testing is rather disregarded among programmers, as well as the knowledge about their effectiveness. Ideally, testing is aimed at generating reproducible results, while following well defined procedures.

While hardware quality assurance often results in quantitative results, same is not the case for software, at least not to the same extent as for hardware. But processes exist for both worlds, to ensure systematic development, as well as quality assurance. Developers of systems integrating both hardware and software have to be aware of their differences. Also, strictly separating the quality measures for software and hardware is not an advisable way to go. The quality properties have to be specified and verified for the complete system and not just its separate modules. The test results of individual modules, usually, can not be superimposed, but the correct behaviour of the whole system has to be demonstrated. Therefore, the deviating aspects of hardware and software quality assurance have to be regarded.

2.2 Terminology and definitions of terms

To clarify regularly used terms, here are definitions in accordance with either [0] or [0]

2.2.1 Quality, Quality requirements, Quality features, Quality measures

- Quality, according to the standard 'DIN 55350 - Concepts for quality management', is defined as: The ability of a unit, or device, to fulfil defined and derived quality requirements.
- Quality requirements describes the aggregate of all single requirements regarding a unit or device.
- Quality features describe concrete properties of a unit or device, relevant for the definition and assessment of the quality. While it does not make quantitative statements, or allowed values of a property, so to say, it very well may have a hierarchical structure: One quality feature, being composed of several detailed sub-features. A differentiation into functional and non-functional features is advised. Also features may have different importance for the customer and the manufacturer. Overarching all these aspects, features may interfere with each other in an opposing manner. As a consequence, maximizing the overall level of quality, regarding every aspect, is not a feasible goal. The sensible goal is to find a trade-off between interfering features, and achieve a sufficient level of quality for all relevant aspects. Typical features, regarding software development include: Safety, security, reliability, dependability, availability, robustness, efficiency regarding memory and runtime, adaptability portability, and testability.
- Quality measures define the quantitative aspects of a quality feature. These are measures, that allow conclusions to be drawn about the characteristics of certain quality features. For example, the MTTF (mean time to failure), is a widespread measure for reliability.

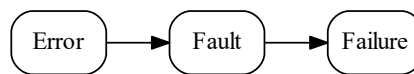


Figure 2.1: Causal chain

2.2.2 Error, Failure, Fault

- Error, the root cause of a device or unit to fail, may originate from operation outside the specification, or from human mistakes in the design.
- Failure, or defect is the incorrect internal state of a unit, and is the result of an error. It exists either on the hard- or software-side and is the cause of a fault, but not necessarily.
- Fault is the incorrect behaviour of the unit, or it's complete cease of service, observable by the user. It is caused by a failure.
- These definitions are in accordance with [0] and [0] and have causal dependencies, depicted in Fig. 2.1.
- While an error can be classified by its persistence, being permanent or transient; Failures and Faults are classified more detailed into consistent/inconsistent, permanent/transient and benign/malign, among other categories.

2.2.3 Correctness

Correctness is the binary feature of a unit or device, loosely described as 'the absence of failures'. A more specific description would be, that a correct software operates consistent to its specification. This implies, that no conclusion about correctness is possible, without an existing specification.

2.2.4 Completeness

Completeness describes, that all functionalities, given in the specification are implemented. This includes normal intended operation, as well as the handling of error-states. It is a necessary, but not a sufficient criterion for correctness.

2.2.5 Testability

Testability describes the property of a unit, to include functionality dedicated only to facilitate the verification of said unit. Supporting concepts include the

- Partitioning of the whole unit into modules, that are testable in isolation. These modules should have little to no side-effects with each other.
- A dedicated debug-unit, making the actual state of the unit observable from outside further assists Testability.
- Another concept is, to specify only as much input space as is necessary, resulting in fewer necessary test-cases to ensure a high coverage.

The aggregate of these concepts is called **design-for-testability**. Generally, time-triggered units support testability to a higher degree, than event-triggered systems.

2.2.6 Safety and Security

- **Safety** means, that a unit is fit for its intended purpose and provides reliable operation within a specified load- and fault-hypothesis.

- **Security**, though, is the resistance of unit against malicious and deliberate mis-usage.

2.2.7 Reliability

Reliability is a dynamic property, giving the probability, that a unit is operational after given time t .

$$\begin{aligned} \text{Reliability ... } R(t) &= e^{-\lambda(t-t_o)} \\ \text{failure rate ... } \lambda &= \frac{1}{MTTF} \end{aligned}$$

An exponential function, decaying from 100% at time = t_0 , where a unit was known to be operating. λ is the failure rate with dimension 'failures/h'

2.2.8 Maintainability

Maintainability is the probability, that a system is repaired and functioning again within a given time after a failure. Note that this includes also the time required to detect the error. A quantified measure for it is the mean-time-to-repair (MTTR).

2.2.9 Availability

Availability combines Reliability and Maintainability into a measure, giving the percentage of time, a unit is operational, providing full functionality.

$$\text{Availability ... } A = \frac{MTTF}{MTTF + MTTR}$$

It is apparent, that a low time-to-repair and a high time-to-failure leads to high Availability.

2.2.10 Robustness

Robustness is actually a property of the specification and requirements. While correctness rates, how far the implemented software complies with the specification, it correct software still might fail in critical situations, that are not covered in the specification. Therefore, to achieve robustness, the specifications have to be examined and ensured, that all critical situations are covered and the expected behaviour of the device under these conditions is defined.

2.2.11 Dependability

Dependability finally, is composed of sufficiently fulfilled levels of

- Reliability
- Availability
- Maintainability
- Safety

...assembled into the common acronym **R.A.M.S.**

2.3 Function-oriented Testing

This chapter establishes methods to design test-cases, to verify a given piece of software against its specification. The first method, named 'equivalence partitioning', assists in reducing all possible inputs to an examined unit, down to a sufficient set of inputs, while the second method 'State based testing' aims to sufficiently cover code, whose behaviour relies heavily on its own condition and history. Both are best suited in a white-box scenario, that means, that the inner structure of the examined software must be known to the tester, for example in form of the not compiled source-code. Equivalence class partitioning might be employed in a black-box scenario, where only a specification is present, but the consequential flaws of such an approach will become apparent in the following chapter.

2.3.1 Equivalence class partitioning

This method is applied most beneficial on a unit- or module level testing. The input- and output spaces of various functions might allow an extreme amount of values, testing them all would lead to unacceptable amount of test-cases and would prevent their execution in a feasible time. Then again, many of those possible inputs would take the same paths through the examined module, in other words, excite the module to the same behaviour. Such a sub-set of inputs forms a common class, a so called 'equivalent class', that can ideally be represented by one input and therefore one test-case. A distinction of cases inside a module would form separate paths for the information to take, therefore form different behaviours of the module itself. Each of those distinctions call for a separate equivalence class and their own test-case. The aggregate of test-cases to cover all possible paths through a unit, or to trigger all possible kinds of behaviour of a unit, form a sufficient set for function-oriented testing. This method, that applies the ancient concept of 'divide and conquer', partitions a unit into low levels of complexity, that can be represented by one single equivalent test-case, thus giving it the name 'equivalence partitioning'.

Equivalence classes should initially be derived from the software's specification and can be distinguished into input- and output-classes, depending of what was specified. While forming a specific output-class it shall be noted, that an according choice of input values has to be defined, presumably exciting the tested unit to the desired or specified output values.

An equivalence class, representing valid input or output values is hence called 'valid equivalence class'. For input or output values, that are specified as invalid, or not specified at all, according 'valid equivalence classes' must be formed as well, to test a units capability in handling those exceptional situations and possibly reveal errors inside a unit. This differentiation in types of test-classes is illustrated in Tab. 2.1.

While output classes are much less common in everyday programming, their importance shall not be neglected: Identical inputs might very well result in different outputs, depending on varying side-effects, that have influence on the inspected unit. This has to be accounted in separate equivalence classes for expected outputs.

Following this first steps of partitioning, the resulting classes shall further be separated into sub-classes that take into account distinction of cases within a module, where

validity-wise	port-wise	
	valid input class	valid output class
	invalid input class	invalid output class

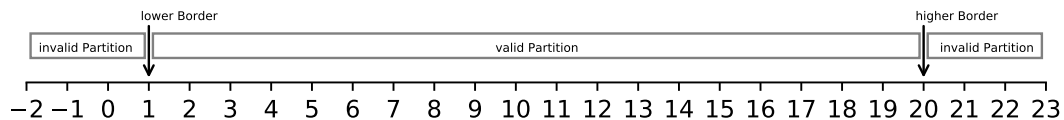
Table 2.1: distinguishing equivalence classes

data might travel several different paths or branches of the source code. This step is only possible in a white-box-scenario, as it affords direct inspection of the source-code. While demanding additional effort, this allows to examine also rather hidden corners of the source-code, that otherwise might go unnoticed and possibly mask hidden errors.

Some examples demonstrate the correct application of the described method:

- valid/invalid input classes:
input is specified as a floating point number between 0 and 20 Volts
→ valid class: $0.00 \leq \text{'test-value'} \leq 20.00$
→ invalid class: $0.00 > \text{test} - \text{value}$ and
→ invalid class: $\text{<test-value>} > 20.00$
- output class:
output is specified for given input filenames as: 0 if file exists, -1 if file does not exist.
→ valid class: Filename of an existing file
→ invalid class: Filename of an inexistent file
→ invalid class: String with a malformed file-path
- dedicated allowed values:
addressed module can be chosen from TriggerA, TriggerB, or TriggerC.
→ valid class: TriggerB
→ invalid class: TriggerK
→ invalid class: Trucker

A visual explanation of the first example is given in Fig. 2.2

**Figure 2.2:** basic equivalence partitions

2.3.2 Boundary value analysis

Until now it might seem, that test-values can be chosen randomly from gathered classes, which is often a sufficient case. But a closely related method called 'Boundary value analysis' refines the selection of test-values. From a set of integers between 10 and 100, with a known code structure to be free from case distinctions, the representative test-value can truly be chosen randomly as 15, 60, or 78. In more complex numerical structures, like floating-point numbers, overarching '0' and negative numbers as input space, a single value becomes insufficient. It is then advisable to deliberately choose values close to the bounding values of a function and in the given case also values close to the '0'. Further explanation of choosing useful values will be given on a slight variation of the first equivalent classes-example: Assumed is a function specified for floating point input values in the range of $\pm 10V$. The given set, visualized in 2.3, has obvious bounding

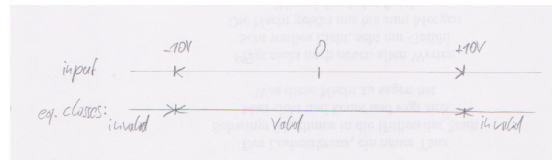


Figure 2.3: equivalent class for composite numerical input

values of ± 10 and -10 , giving the first to test-values. Small values deviating from $\pm 10V$ are also values to test for. Furthermore, '0' and small values deviating from 0 in positive and negative direction will reveal the units stability, in case, the input is used as a divisor. **To-Do: saubere Grafik wie 2.2**

→ valid classes: $+10V$, $+9.99V$, $-10V$, $-9.99V$, $0V$, $+0.01V$, $-0.01V$

→ invalid classes: $+10.01V$, $-10.01V$

Every value has to be applied via a separate testcase, to alleviate which values cause problems, in case of failing tests.

Boundary value analysis and equivalence class partitioning are closely related and often mentioned in unison, nevertheless, their separate description in this chapter is intended to specify their different applications.

2.3.3 State based testing

2.4 Coverage metrics

This chapter describes dynamic testing techniques that assess the completeness of the test based on the coverage of the software source code. Coverage describes the amount of source code, that is executed during testing of the examined software. Therefore, they are referred to as structure-oriented testing techniques. The techniques described are based on the control structure or the control flow of the software to be tested. For this reason one speaks of control flow-oriented, structure-oriented test techniques. This group of

test techniques is of great practical importance. This applies in particular to their use in module testing, the so-called 'testing on a small scale'. The group of control flow oriented testing techniques is well supported by test tool vendors. In addition, there are accepted minimum criteria in the field of control flow-oriented tests that should be considered in terms of an adequate test. As minimal, the so-called branch coverage test is a necessary acceptance test procedure. In particularly critical areas of application, relevant standards require more extensive tests, for example a so-called condition coverage test. Certain control-flow-oriented test techniques are of such a fundamental nature that an examination that does not use these techniques, particularly in the module test, must be rated as insufficient.

2.4.1 Properties and goals

Because control flow-oriented test techniques belong to the group of structure-oriented test techniques, they have their respective advantages and disadvantages. Test completeness is assessed based on coverage of the control structure or control flow. A corresponding specification is required for the assessment of the expenditure. Like all structure-oriented test techniques, control flow-oriented test techniques do not define any rules for the generation of test cases. It is only important that the test cases cause corresponding coverage in the structure. This degree of freedom in test case generation is extremely important, as it allows other test case generation techniques to be combined with structure-oriented testing techniques.

The most important area of application of the control flow-oriented test techniques is the unit test. Control flow-oriented test techniques can still have a certain importance in integration testing, while they are not used in system testing. Control flow oriented testing techniques look at the structure of the code. In particular, aspects of the processing logic that are represented in the software as instructions, branches, conditions, loops or paths are considered. The disadvantage is that this approach is blind to omission errors in the software. Unrealized but specified functions are only recognized by chance, there is no code to test for these functions.

The test basis is the so-called control flow graph. The control flow graph can be created for any program implemented in an imperative programming language. For this reason, the control flow-oriented test techniques can be applied equally to all programs created in an imperative language via the representation of a software module to be tested as a control flow graph. Tools to support control flow-oriented testing techniques usually generate control flow graphs, as portrayed in fig. 2.4, and use them to represent the test results.

To-Do: Günther: bitte bis hier durchlesen

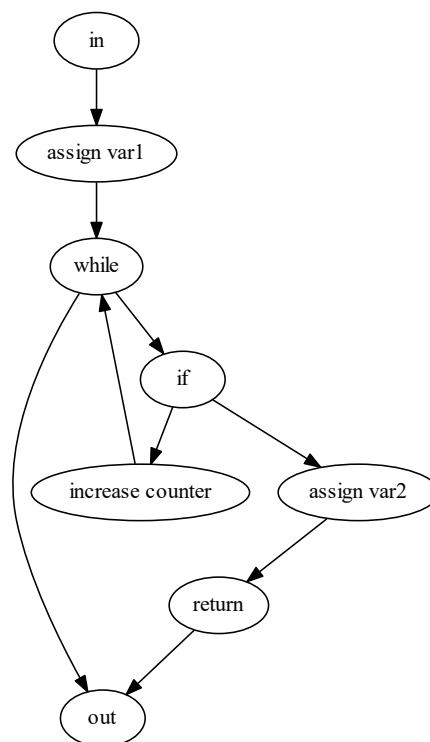


Figure 2.4: exemplary control flow graph of a looped function

References

Literature

- [0] Hermann Kopetz. *Real-time systems - design principles for distributed embedded applications*. Vol. 395. The Kluwer international series in engineering and computer science. Kluwer, 1997 (cit. on pp. 8, 9).
- [0] Peter Liggesmeyer. *Software-Qualität - testen, analysieren und verifizieren von Software*. Spektrum Akadem. Verl., 2002 (cit. on pp. 7–9).