Otto-Friedrich University Bamberg

Lehrstuhl für Softwaretechnik und Programmiersprachen

Bachelor's Thesis

Im Studiengang Software Systems Science
der Fakultät Wirtschaftsinformatik und Angewandte Informatik
der Otto-Friedrich Universität Bamberg

Zum Thema

Test Cost Estimation of Model-based Embedded Software

Vorgelegt von:

Frank Kaiser

Themensteller:
Prof. Dr. Gerald Lüttgen

Abgabedatum:
12.10.2018

Contents

1 Introduction 1

2 Background 2
2.1 Glossary 2
2.2 SIBAS Control Technology 2
2.3 SAMAPI - Sibas Application Model API Specification 3

3 Approach 4
3.1 Halstead-Metric 4
3.2 Cyclomatic-Complexity 5
3.3 Custom Approach: Signal Depth Analysis 5
3.4 Implementation: sbtSWAT 6

4 Evaluation 8

5 Related Work 11
5.1 Empirical Study of Software Metrics 11
5.2 Calculation and Visualization of Model Complexity in Model-based, Safety-relevant Software 11

6 Future Work 12
6.1 Integrating into Toolchain 12
6.2 Time Cost Estimation 12

7 Conclusion 13

List of Figures

1 SAMAPI - Class Overview 3
2 Function Group example 4
3 sbtSwat User Interface 6
4 Function ID of Function Group KONSTLF0.010 7
5 Euclidean distance - Function ID approach 8
6 Euclidean distance - Function Group approach 9
7 Complexity comparison NSB - Function ID approach 9
8 Complexity comparison NSB - Function Group approach 10
9 Complexity comparison BLE - Function ID approach 10
10 Complexity comparison BLE - Function Group approach 10

1 Introduction

Quality control in software development is a fundamental step to a reliable product. Especially in safety critical area, software is not allowed to malfunction. However, the longer tests take, the more expensive the products get. If you know the test cost of your product, you can coordinate man hours in the most effective way, minimizing costs while keeping the quality high.

In this thesis a tool is developed to analyze model-based software in order to estimate a test cost. In the process three different approaches emerged. The Halstead Software Metrics subsection 3.1, McCabe's cyclomatic complexity subsection 3.2 and a newly introduced custom approach subsection 3.3. Tp evaluate all approaches experts reviewed the model-based software, grading it on a scale from 0 to 10 according to its complexity to understand section 4.

2 Background

This section establishes the terminology used in this paper and gives an overview of the terminology used in other works.

2.1 Glossary

This section briefly explains terms used in the SIBAS G programming language.

Program The complete SIBAS program that can be compiled and uploaded to run cyclical on its device. It is hierarchic ordered into function packages.

Function Package A function package describes a broader functionality of a component, e.g. the parking brake, and consists of multiple function groups.

Function Group Function groups are the smallest part of the program hierarchy. They can either contain multiple function groups or models describing the software. The model itself consists of two entities: function blocks and signals.

Function Block Function blocks are predefined methods that can take multiple inputs and have multiple outputs.

Signal The variables of the program are projected as signals. Visually they are either represented by a flag when they don't origin in the current function group or otherwise as a line between two function blocks. There are different types of signals. For this context most importantly, global signals and local signals. Global signals origin from another function package or are created to be read from different function packages. They are only allowed to be written to by one actor and can be identified by an '$' in front of the signal name. Local variables are only write and readable from the same function package.

Function ID Function IDs are unique identifiers that are assigned by the developer to function groups that implement a common function. A function ID consists of at least one function group, though more complex functions are spread over multiple IDs.

2.2 SIBAS Control Technology

SIBAS 32, the successor of SIBAS 16, is a control technology used in trains. Its goal is to provide a common interface between different components, e.g. central control device or brake control device. SIBAS 16 was first introduced in 1983 and was further developed until in 1992 SIBAS 32 was released. The software that runs SIBAS is projected in an editor called SIBAS G in a model-based approach. The model translates automatically into C code and eventually compiles to run as intended on embedded systems. Every function block is validated before being allowed to use, hence when testing the software, only the logic of the software needs to be tested.

2.3 SAMAPI - Sibas Application Model API Specification

SAMAPI provides the possibility to read or create SIBAS models. It is written in Java but is distributed to use in Python 2.7 via Jython[1].

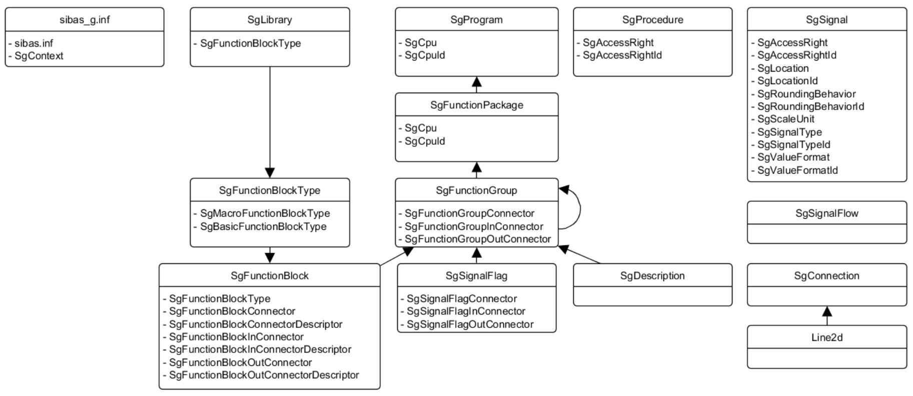


Figure 1: SAMAPI - Class Overview

Figure 1 describes the structure of the API. First, either a function package file or a program file has to be loaded, returning a SgFunctionPackage or SgProgram object respectively. On this object several methods can be called, e.g. 'getFunctionBlocks' or 'getFunctionGroups', returning a set of specified objects. Whenever calling a function on an object, ot only returns objects that are in its context.

[1]http://www.jython.org/docs/index.html

3 Approach

In this section the general approach is described. To estimate the test cost, three different approaches are implemented: The Halstead metric, cyclomatic complexity and a custom solution that considers the testing method and the properties of the underlying simulation software. For each approach, the calculation was done on a function group basis and using function IDs.

3.1 Halstead-Metric

Halstead defines multiple metrics to estimate the complexity of software [1]. For this use-case the most important one is the difficulty, which is defined as $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$. The volume $V = N \times \log_2 n$ will additionally be evaluated in section 4. All Halstead metrics are based on four parameters:

- n_1 = amount of distinct operators
- N_1 = total number of operators
- n_2 = amount of distinct operands
- N_2 = total number of operands

Function blocks are interpreted as operators, the inputs of a function group as operands. For the function group in Figure 2 this results in $n_1 = 2$, $N_1 = 3$, $n_2 = 6$ and $N_2 = 6$. Note that 'LFRDYA' does not count as operand because it is build inside this function group. Using these parameters the software vocabulary $n = n_1 + n_2$ and the length $N = N_1 + N_2$ can be calculated.

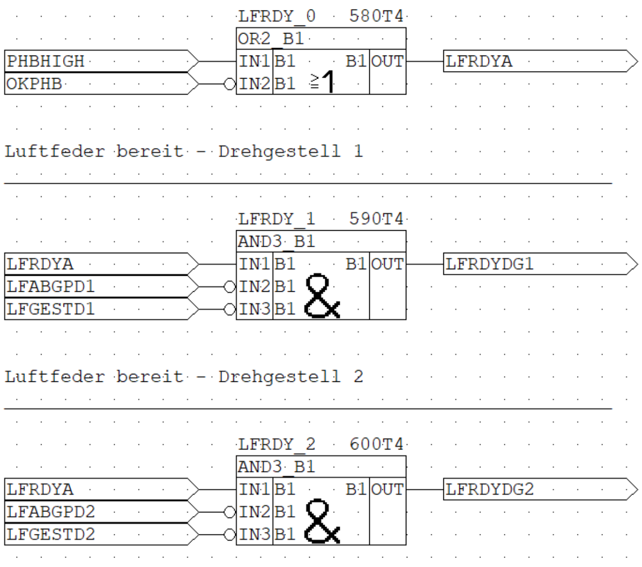


Figure 2: Function Group example

3.2 Cyclomatic-Complexity

Cyclomatic complexity is a graph-theoretic complexity measure introduced by McCabe [3]. He defines the cyclomatic number $v(G) = e - n + 2p$ of a graph G, where n is the number of vertices, e the amount of edges and p the number of connected components. When looking for connected components only the specific function ID or function group is considered. This decision was made for two reasons. First, the algorithm to find all connected components takes quite a long time when considering the whole program, secondly the result will mostly always be p = 1. For the function group in Figure 2 the result would be $v(G) = 11 - 3 + 2 * 1 = 10$. $p = 1$, since the function blocks 'LFRDY_1' and 'LFRDY_2' are connected via the signal 'LFRDYA' to 'LFRDY_0'.

3.3 Custom Approach: Signal Depth Analysis

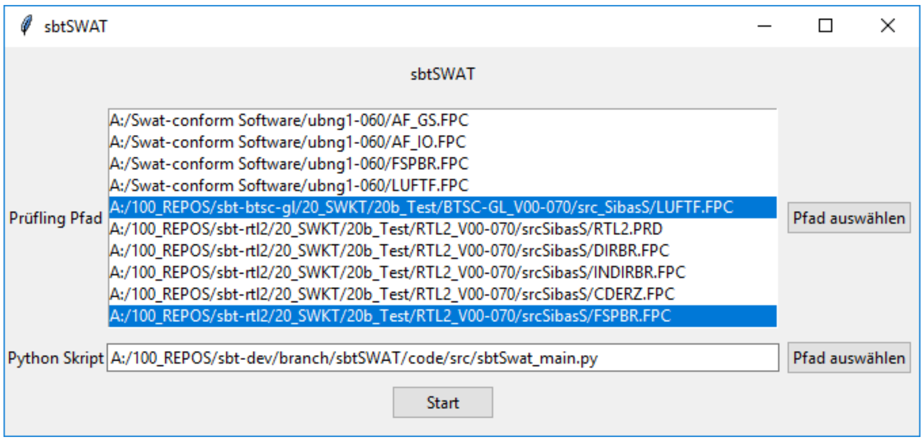
When testing the software, it is currently only possible to manipulate global signals (global variables) of a function package, as local signals are overwritten each cycle in the simulation. Hence, the longer the path from the signal that shall be validated to its global signals is, the more complex it is to understand which values produce a specific result. F.e when considering the function group in Figure 2, the outgoing signals are 'LFRDYA', 'LFRDYG1' and 'LFRDYG1'. For each of these signals Algorithm 1 is executed, that takes the first function block (fb) before the signal as parameter and counts the visited function blocks along the path until a global signal is reached. The resulting difficulty for that function group is defined as: $D = \log_2(S_a * g * G)$, where S_a = the average signal depth of all outgoing signals, g = number of unique global signals and G = the number of all global signals needed. All three parameters are equally weighted in this function because it is not obvious which parameter has the biggest impact on complexity. The logarithmic function is applied to the product to prevent linear scaling. This is inspired by Stürmer et. al, which concluded similar for the amount of inputs and function blocks [4].

Algorithm 1 Recursive signal depth algorithm

```
1: procedure CALCULATESIGNALDEPTH($fb$)
2:     $signalDepth \leftarrow 1$
3:     $fb.visited \leftarrow True$
4:     for signal in fb.InSignals do                         ▷ InSignals is empty
5:         if signal.isGlobal() then
6:             $globalSignals \leftarrow signal$         ▷ Add to list of all global signals
7:             return signalDepth
8:         end if
9:         $beforeFb = signal.getSource()$
10:        if NOT beforeFb.visited then                      ▷ cycle prevention
11:            $signalDepth \leftarrow signalDepth + calculateSignalDepth(beforeFb)$
12:        else
13:            $signalDepth \leftarrow signalDepth + 1$           ▷ Stop recursion on cycle
14:        end if
15:    end for
16:    return $signalDepth$
17: end procedure
```

3.4 Implementation: sbtSWAT

The tool which implements this approach is called sbtSWAT. An acronym for Siemens brake testing software analysis tool. It consists of six python2 modules: sbtSwat_main.py, halstead.py, cyclomaticComplexity.py, signalDepth.py, functionIdTracer.py and dataHandler.py. To start programs using the SAMAPI a batch file that setups the Jython Environment and loads the API has to be called. In this batch file the python script needs to be specified with a path to a program or function package as command line argument. In order to ease the execution of sbtSwat a minimalistic user interface was developed in Python3.7 (see Figure 3). Here, Python3 was used to allow easier integration in the currently employed toolchain.


sbtSWAT

sbtSWAT

Prüfling Pfad

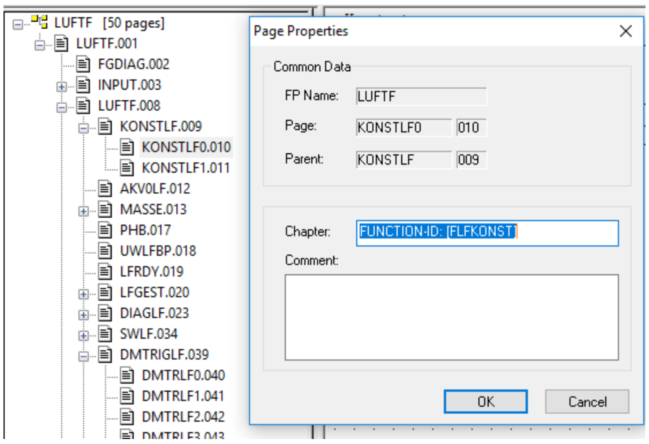
A:/Swat-conform Software/ubng1-060/AF_GS.FPC
A:/Swat-conform Software/ubng1-060/AF_IO.FPC
A:/Swat-conform Software/ubng1-060/FSPBR.FPC
A:/Swat-conform Software/ubng1-060/LUFTF.FPC
A:/100_REPOS/sbt-btsc-gl/20_SWKT/20b_Test/BTSC-GL_V00-070/src_SibasS/LUFTF.FPC
A:/100_REPOS/sbt-rtl2/20_SWKT/20b_Test/RTL2_V00-070/srcSibasS/RTL2.PRD
A:/100_REPOS/sbt-rtl2/20_SWKT/20b_Test/RTL2_V00-070/srcSibasS/DIRBR.FPC
A:/100_REPOS/sbt-rtl2/20_SWKT/20b_Test/RTL2_V00-070/srcSibasS/INDIRBR.FPC
A:/100_REPOS/sbt-rtl2/20_SWKT/20b_Test/RTL2_V00-070/srcSibasS/CDERZ.FPC
A:/100_REPOS/sbt-rtl2/20_SWKT/20b_Test/RTL2_V00-070/srcSibasS/FSPBR.FPC

Python Skript A:/100_REPOS/sbt-dev/branch/sbtSWAT/code/src/sbtSwat_main.py

Figure 3: sbtSwat User Interface

It allows the user to select one or multiple paths and then modifies and calls the batch file according to user inputs, giving feedback to the user depending on the exit status of the batch process. On quit, all settings are saved in a serialized python file created by the pickle[2] module. When a program is loaded, sbtSwat_main gathers all function packages included and proceeds with the same procedure as when a function package was specified.

[2]https://docs.python.org/3/library/pickle.html



```
LUFTF  [50 pages]
  LUFTF.001
    FGDIAG.002
    INPUT.003
    LUFTF.008
      KONSTLF.009
        KONSTLF0.010
        KONSTLF1.011
      AKVOLF.012
      MASSE.013
      PHB.017
      UWLFBP.018
      LFRDY.019
      LFGEST.020
      DIAGLF.023
      SWLF.034
      DMTRIGLF.039
        DMTRLF0.040
        DMTRLF1.041
        DMTRLF2.042
        DMTRLF3.043
```

Page Properties

Common Data

FP Name:	LUFTF	
Page:	KONSTLF0	010
Parent:	KONSTLF	009

Chapter: FUNCTION-ID: [FLFKONST]

Comment:

Figure 4: Function ID of Function Group KONSTLF0.010

Before finding parameters for the listed approaches above, all functionIDs with their respective function groups are traced. The result is a key-value pair (<fID>,<[fg]>). Each module representing an approach specifies a 'calculate' method that takes a list of function groups as parameter and then return the calculated result. This allows to use the same function call for the function ID and the function group approach.

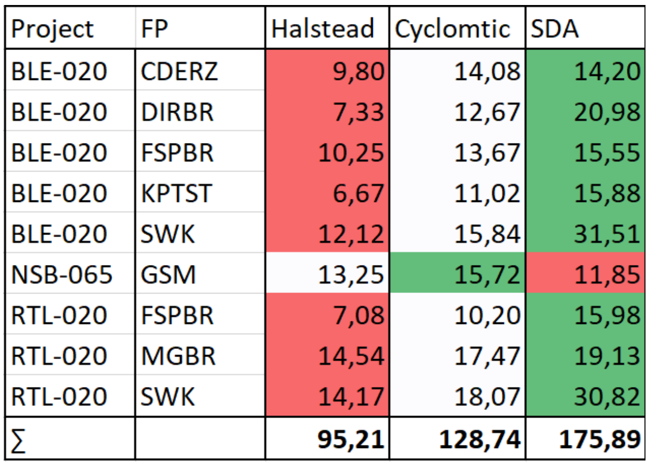
4 Evaluation

In order to evaluate these approaches, the resulting values are normalized to a scale from zero to ten (Equation 1, though zero is only the result if the value could not be computed, e.g. when resulting parameters would divide by zero. Additionally, test engineers were conducted and rated different projects using the same scale. Eventually, the euclidean distance (see Equation 2) is used to calculate the distance of each data set in regard to the manually conducted approach.

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}} * 10 \tag{1}$$

$$d(p,q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2} \tag{2}$$

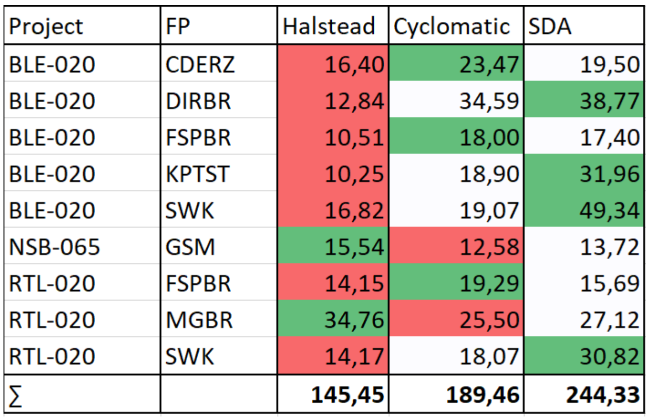
In total, ten function packages of three projects were evaluated by three different experts. For function IDs, the results for the euclidean distance of each approach to the experts estimation can be seen in Figure 5. For the approach by function groups Figure 6 shows the results. The values marked red are the lowest and therefore the closest to the experts reference, while green marks the biggest number, which indicates a large difference in values. For both testing methods (function ID and function group) the Halstead difficulty is performing best with having the smallest euclidean distance in 17 of 20 cases. Cyclomatic complexity showed to be right in the middle between the signal depth analysis (SDA) and the Halstead difficulty.



Project	FP	Halstead	Cyclomtic	SDA
BLE-020	CDERZ	9,80	14,08	14,20
BLE-020	DIRBR	7,33	12,67	20,98
BLE-020	FSPBR	10,25	13,67	15,55
BLE-020	KPTST	6,67	11,02	15,88
BLE-020	SWK	12,12	15,84	31,51
NSB-065	GSM	13,25	15,72	11,85
RTL-020	FSPBR	7,08	10,20	15,98
RTL-020	MGBR	14,54	17,47	19,13
RTL-020	SWK	14,17	18,07	30,82
∑		**95,21**	**128,74**	**175,89**

Figure 5: Euclidean distance - Function ID approach

In one case SDA performed best in the function ID approach. In Figure 7 the three examined approaches and the expert evaluation are plotted for NSB 065 - GSM. Function ID 4 is one of the few cases, where the experts evaluation is higher than all approaches. Though when studying all data sets, SDA scores usually the highest overall. This may be one of the reasons, why SDA performed worst in most cases, as the experts rated mostly between two and six. Though



Project	FP	Halstead	Cyclomatic	SDA
BLE-020	CDERZ	16,40	23,47	19,50
BLE-020	DIRBR	12,84	34,59	38,77
BLE-020	FSPBR	10,51	18,00	17,40
BLE-020	KPTST	10,25	18,90	31,96
BLE-020	SWK	16,82	19,07	49,34
NSB-065	GSM	15,54	12,58	13,72
RTL-020	FSPBR	14,15	19,29	15,69
RTL-020	MGBR	34,76	25,50	27,12
RTL-020	SWK	14,17	18,07	30,82
∑		**145,45**	**189,46**	**244,33**

Figure 6: Euclidean distance - Function Group approach

all approaches seem to have similar tendencies, since the plots changes mostly in the same directions. It seems the biggest difference between all approaches is the scale. General more complex function packages may be better analyzed by SDA, while more common ones using Halstead.

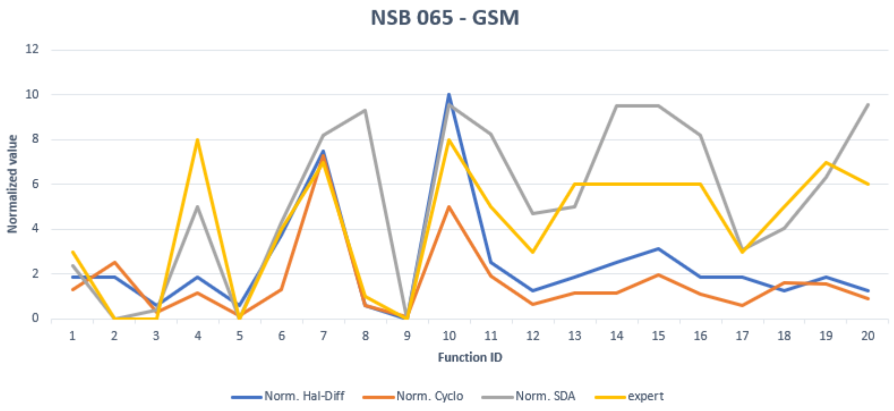


Figure 7: Complexity comparison NSB - Function ID approach

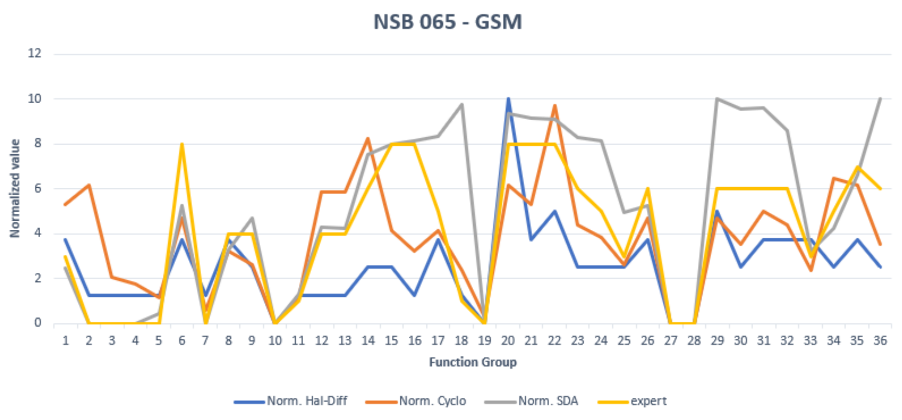


Figure 8: Complexity comparison NSB - Function Group approach

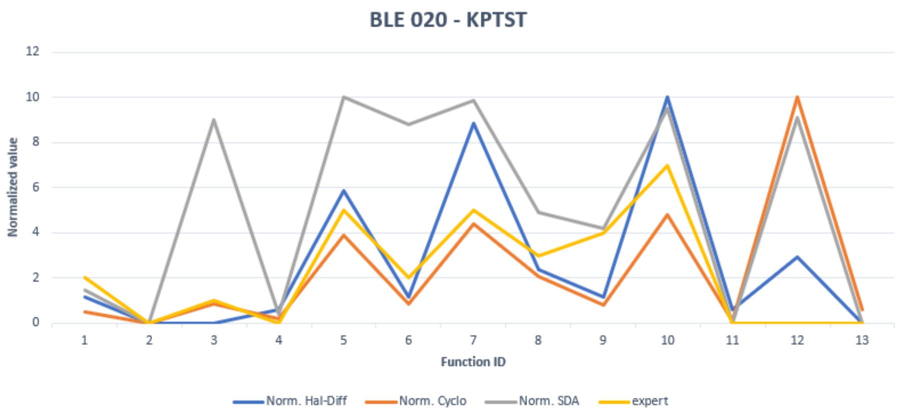


Figure 9: Complexity comparison BLE - Function ID approach

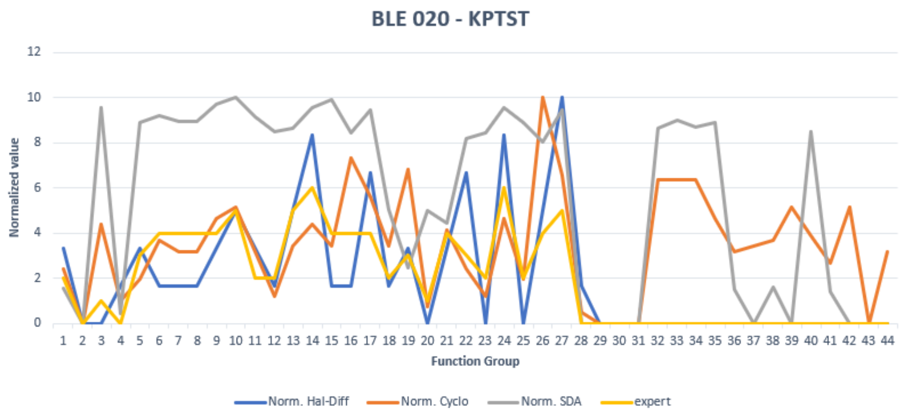


Figure 10: Complexity comparison BLE - Function Group approach

5 Related Work

5.1 Empirical Study of Software Metrics

In 1987 Li and CHEUNG published their empirical study of software metrics [2]. They wrote a static Fortran source code analyzer to automatically analyze 255 different programs (student assignments) and compared 31 different metrics including a new introduced metric. They divide these static measures in three categories: Volume, measuring the size of a product, Data Organization, measuring usage and visibility of data, and Control Organization, measuring the comprehensibility of control structures. Halstead metrics would be in the first category, while McCabe's cyclomatic complexity fits the second one. LI and Cheng propose to first use Halstead metrics to put the analyzed software in different categories and then use cyclomatic complexity to a more fine-grained estimation in each category. He concludes that the validity of any tested measure cannot be asserted with precision and that a metric is valid if it succeeds to reflect what it it meant to measure.

5.2 Calculation and Visualization of Model Complexity in Model-based, Safety-relevant Software

Stürmer et al. estimate complexity of model-based software by applying the Halstead metrics on a model [4]. They argue against cyclomatic complexity in their usecase as the model complexity is not significantly affected by control flow, but by the amount of blocks and how they are connected with their corresponding signals. As parameters used for the Halstead metrics they chose mostly similar values as shown here in subsection 3.1, though for N_2 they decided to choose the amount of outgoing signals for a function block.

6 Future Work

6.1 Integrating into Toolchain

The sbtSWAT functionality shall be integrated into the current tool in use that allows test automation.

6.2 Time Cost Estimation

One of the reasons the test cost of software is established, is to help planning the test order. If a reasonable time frame can be derived without having to examing the software, project planners will have a easier time meeting the deadline to a software release. In order to achieve a good time estimation, the time needed for every function group needs to be noted and put in relation to the estimated complexity values.

7 Conclusion

In this thesis three different approaches to estimate the test cost of model-based embedded software, the Halsted metric, McCabe's cyclomatic complexity and a newly introduced solution tailored to system specifics were implemented and then evaluated with the estimations of experts that test the software everyday. Halstead's software metric for the difficulty to understand software was in an overwhelming amount of cases the closest to the experts estimation. Though that does not mean the other metrics are bad, just that they are designed give more weight on different specifics.

References

[1] Halstead, M.H.: Elements of Software Science (Operating and Programming Systems Series). Elsevier Science Inc., New York, NY, USA (1977)

[2] Li, H.F., Cheung, W.K.: An empirical study of software metrics. IEEE Transactions on Software Engineering SE-13(6), 697–708 (June 1987)

[3] McCabe, T.J.: A complexity measure. IEEE Transactions on Software Engineering SE-2(4), 308–320 (Dec 1976)

[4] Stürmer, I., Pohlheim, H., Rogier, T.: Berechnung und visualisierung der modellkomplexität bei der mo- dellbasierten entwicklung sicherheitsrelevanter software (Jan 2010)

Eingeständigkeitserklärung

Ich erkläre hiermit gemäß §17 Abs. 2 APO, dass ich die vorstehende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Bamberg, den 11. Oktober, 2018 Frank Kaiser