Altinger Harald

State of the Art Software Development in the Automotive Industry and Analysis upon Applicability of Software Fault Prediction

Doctoral Thesis

Graz University of Technology

Institute for Softwaretechnology
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Supervisor: Univ.-Prof. Dipl-Ing. Dr.techn. Franz Wotawa
Graz, Univeristy of Technology
Austria
Evaluator: Associate Professor Shuji Morisaki
Nagoya University
Japan

Graz, November 2016

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.

Graz, ____________________ ______________________________

Date Signature

Eidesstattliche Erklärung[1]

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, ____________________ ______________________________

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

To the love of my life, Katharina.

To my parents, who have supported me during my whole life.

Special thanks goes to Prof. Wotawa, who supervised this thesis and has always been helpful with the reviewing and discussing of my work and co-authoring of my publications.

Further thanks goes to my publications co-authors Yanja and Steffen for their supportive help and long discussions.

Additional thanks goes to Clair who performed the first review and marked spelling and formulation errors.

Abstract (English)

In recent years the amount of software within automobiles has increased up to 100 Million Lines Of Code in modern day premium vehicles. Virtually all innovations in automotive engineering in the last decade include software components. Parallel to this increasing amount, testing becomes more vital. Automotive software development follows restrictive guidelines in terms of coding standard, language limitations and processes. Traditionally testing is a core part of automotive development, but the raising number of features increases the time and money required to perform all tests. Repeating them multiple times due to programming errors might jeopardises a cars introduction on the market. Software Fault Prediction is a new approach to forecast bugs already at time of commit, thus to guide test engineers upon defining testing hotspots. This work reports on the first successful application using model driven and code generated automotive software as a case study and a success prediction rate up to 97% upon a bug or fault free commit. A compiled and published dataset is presented along with analysis upon the used software metrics. Performance data achieved using different machine learning algorithms is given. An indepth analysis upon factors preventing Cross-Project Fault Prediction is conducted. Further usage and practical application areas will conclude the work.

Zusammenfassung (Deutsch)

Der Anteil an Software im Automobil hat in den vergangen Jahren deutlich zugenommen und umfasst in modernen Oberklassenfahrzeuge bis zu 100 Millionen Codezeilen. Nahezu alle Innovationen des Automobilbaues im letzten Jahrzehnts beinhalten Software Komponenten. Einhergehend mit dieser Zunahme wird das Freitesten jener Komponenten wesentlich wichtiger. Die Softwareentwicklung in der Automobilindustrie ist sehr strengen Entwicklungsrichtlinien in Form von Codingstandards, Einschränkungen im Funktionsumfang der Programmiersprache sowie Prozessvorschriften unterworfen. Traditionell ist das Testen eine Kernaufgabe der Automobilentwicklung, jedoch mit der zunehmenden Anzahl an Softwarefunktionen steigt sowohl der monitäre als auch zeitliche Aufwand alle Systeme frei zu testen. Müssen Tests mehrfach aufgrund von Programmierfehlern wiederholt werden kann dies die Markteinfürung eines neuen Modelles gefährden. Die Software Fehler Vorhersage ist eine neue Methode um Fehler bereits zum Commit Zeitpunkt vorherzusehen und damit Test Ingenieuren eine Möglichkeit zu bieten ihre Testschwerpunkte zu definieren. Dies ist die erste Arbeit zur Vorhersage von Fehlern welche Model basiert entwickelte und automatisch Code generierte Software als Fallstudie verwendet. Bei 97% der commits kann korrekt zwischen fehlerfrei und fehlerbehaftet unterschieden werden. Das erstellte und veröffentlichte Datenset wird gemeinsam mit einer Analyse zu den verwendeten Software Metriken und den erreichten Genauigkeiten unterschiedlicher maschineller Lernmethoden beschrieben. Im Anschluss erfolgt eine genaue Untersuchen der Ursachen die zur nicht Anwendbarkeit der projektübergreifenden Fehlervorhersage führen. Abschließend werden weitere Anwendungsfelder für die praktische Anwendung des Systems präsentiert.

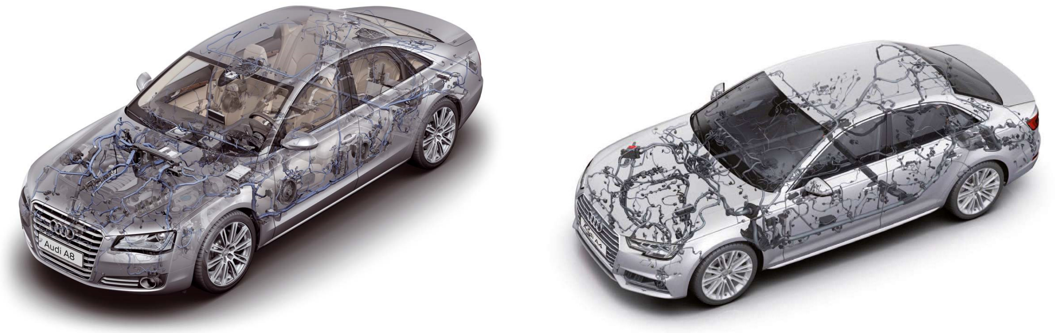
Contents

1. Introduction

This chapter will start with a general motivation 1.1 to automotive software engineering as a computer science research field, continuous with the identified problems 1.2 and research questions on software testing within the automotive industry, presents an overview to the thesis contribution 1.3 and concludes with the thesis organization 1.4.

1.1. Motivation

Within the last decade **Software gained importance in cars**. A modern day premium car, *example given* the 2015 Audi A4 [1] may be equipped with up to 90 Electronic Control Unit (ECU), two high resolution displays, two Subscriber Identification Module (SIM) cards, 11 communication networks (Controller Area Network (CAN), FlexRay, Media Oriented Systems Transport (MOST)) and up to six antenna systems (radio, Keyless Entry Start and Exit System (Kessy), WiFi, *etc.*) ensuring wireless communication between the car and various infrastructure. From a computer scientist's perspective a modern day car is a heterogeneous network of embedded computers performing local and distributed tasks. In addition to transport capabilities customers demand up to date entertainment (including music, video or online streaming) and comfort(climate control, massage seats, *etc.*) in a modern day car. Various features, *example given* Advanced Driver Assistance Systems (ADAS), rely on data fusion between multiple sensors and pre calculated values on various ECU. A wide range of sensors starting from simple switches or rotary encoders to advanced Global Positioning System (GPS) Antennas or Radar Sensors will be used to sense the car's environment or interact with the driver. Realizing innovative ADAS like Adaptive Cruise Control (ACC) or Matrix headlamps requires fusioning pre processed measurement data from a camera sensor and a radar sensor as well as a lookup from the road traffic database. This requires four ECU to



(a) Audi A8 Electric wires diagram. Picture extracted from [3]

(b) Audi A4 Electric wires diagram. Picture extracted from [1]

Figure 1.1.: Audi A4 and A8 schematic showing the electric wiring harness

work together in realizing one specific driving function. Figure 1.1 shows two modern day premium cars with all their electronic systems and wires, which sum up to 2.5 km in total length, leaving a modern day car to be one of the most technical places within a humans daily live. Following Pretschner *et al.* [2] a 2007 BMW 7 Series contained about 270 software based functions and 67 ECU powered by 65 megabyte of data, the 2010 Model has been expected to contain one gigabyte of software.

The amount of software required to operate a car or an aircraft has increased during past years, even putting the automotive domain in the lead. Robert Charette [4] compares the complexity between an aircraft and a car demonstrating that it requires more Lines Of Code (LOC) to operate a typical car than an aircraft. Dvorak *et al.* [5] states that software realized functionality within a military aircraft raises from below 10% on an 1960is F-4 to 80% on a modern day F-22. Similarly the author states a representative car from General Motors (GM) rises from 100.000 LOC within the 1970is up to 1.000.000 LOC in 2010. Broy [6] states modern day premium cars can contain up to 100.000.000 LOC. In previous works Broy [7] reveals **electronics and software development consumes up to 40% of the whole development budget** nowadays. This is in line with a forecast by Siemens released in 2005, see Aschenbrenner [8]. Similarly a market research from Ehmer [9] states that 20% of the car's total development budget in 2000 will increase to 35% by 2010. They will be distributed among 2% basic software, 8% operating system and 28% application software.

A car recall costs money. Whenever a malfunction within a car is a threat to human life, the law forces an Original Equipment Manufacturer (OEM) to perform repairs. In terms of software this means developing a hotfix and distributing it. Most OEM do not equip their cars with over the air update capabilities this means the car has to be moved into a workshop. Within recent years there have been multiple recalls, compare Figure 1.2. The total number can be extracted by querying the National Highway Traffic Safety Administration (NHTSA) database [10], the amount of software related recalls can been extracted counting each entry containing the terms ”software” or ”program” within the recall description or required repair action. The number of sold vehicles can be gathered from Wards Auto [11]. The majority of recalls were due to mechanical deficits, but the share of software related recalls is increasing. Within the automotive industry such a recall can cost millions, as an OEM has to pay for contacting the customer, maybe a rental car and the workshop to replace the software. In addition, National Automobile Dealers Association (NADA) published a Whitepaper [12] analysing the impact of recalls on a cars (retail)value. They recognized an increasing number of affected vehicles by recalls within the last decade. Analysing an OEM's average car price compared to competitors they found clear impacts of a recall on achievable market prices, *example given* Toyota dropped by -20% after the 2009 recall on self accelerating models. The authors conclude avoiding a recall will be of economic interest.

Finding bugs later costs more money. A recent Whitepaper from Klocwork [14] stated that finding bugs in early development phases might cost 25$, in a later phase this could climb up to 16.000$. The authors values are based on standard software. Tassey [15] presents multiple analysis concerning the costs when finding bugs in different development stages. The author states 70% of all errors are introduced during the requirements phase but 50% of all bugs will be discovered during the integration testing phase. Further the authors analysed the cost of fixing bugs, see Figure 1.3. In line

[1] Model-year is the first year where a car type is introduced. If a recall is reported in 2008 and 2010, but the car was lunched in 2004, the recall will always bee counted for Model-year 2004. One car can be affected by multiple recalls during its lifetime. The recall data is available via NHTSA [10], the sale statistic via Wards [11].

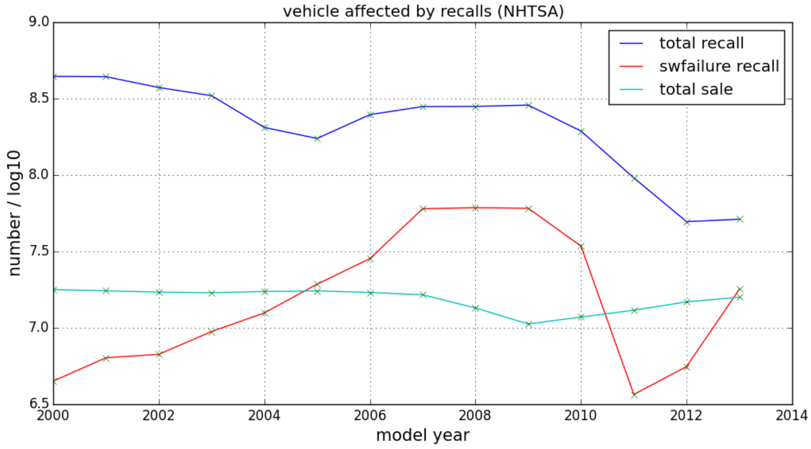


Figure 1.2.: NHTSA recall statistics on Model-year 2000 - 2014, comparing recalls in total with software caused recalls and sales [1]. One can see more recalls than sold vehicles within multiple years. The graphic has been extracted from Altinger *et al.* [13].

with Capers [16] the later a defect is detected, the more expensive it is to fix it. In terms of the automotive industry the numbers will be even higher, as the law forces antecedent tests to be repeated. The majority of tests requires expensive hardware and personnel *example given* to ride prototype cars for a defined mileage.

Within a software's Product Life Cycle (PLC), maintenance can cause the highest costs. ISO/IEC 14764 [17] defines software maintenance as the modification of a product after delivery to a customer. The aim of such a modification is to correct faults, to improve performance or to adopt other quality attributes of the product. As analysed by Kozlov *et al.* [18] 49% to 75% of the total software costs are caused by maintenance. Kozlov *et al.* examined data between the 1970s and 1990s. Recent data published by Confora *et al.* [19] indicate even more than 80% of the total PLC costs are caused by maintenance nowadays. Shull *et al.* [20] states that fixing a software fault during maintenance caused higher costs than finding and fixing it during the early phase of the software's PLC, which is in line with Capers [16]. Even if concrete figures vary, Shull *et al.* [20] analysed that

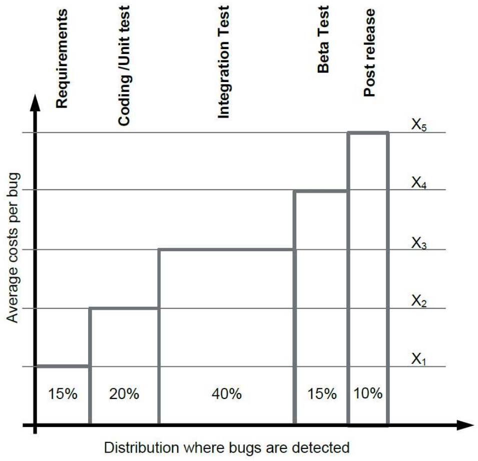


Figure 1.3.: Costs to fix detected bugs, the graphic has been adopted from Tassey [15].

the effort increase by 100:1 for critical defects on large projects and 2:1 for non-severe bugs discovered after release.

Summarizing this chapter's arguments, the share of software increases, fixing bugs out in the field (after release) costs huge amount of money and has a negative impact on the OEM's reputation. Thus finding bugs during early development stages is of economic interest.

1.2. Problem Statement

As stated in the previous chapter, developing software is a complex and expensive task. According to Broy [6] and Aschenbrenner [8] 30-50% from a car's total development costs will be dedicated to software by 2030. Testing has always been a core part of Automotive engineering, as the W-Development process defines a testing stage for every development stage, compare Jin-Hua *et al.* [21]. Indicators where to particular spend TestCase (TC) are welcome to increase efficiency in testing.

The following research questions are identified:

RQ1: What are the common tools automotive software engineers use to specify requirements and write their software?

RQ2: Is it possible to use fault prediction within automotive software projects?

- **RQ 2.1:** Does fault prediction benefit from restrictive development guidelines (Coding-standards and development processes)?
- **RQ 2.2:** What are influential parameters for fault prediction to performing usefully?
- **RQ 2.3:** Do (re)sampling strategies influence the achievable performance?
- **RQ 2.4:** Is it possible to establish Cross-Project Fault Prediction (CPFP) within the restrictive development settings?

RQ3: Which metrics perform best for generated code within automotive software?

- **RQ 3.1:** Which metrics are independent and share no correlation with others?
- **RQ 3.2:** Do metrics represent the occurred bugs?

RQ4: What are good fault predicting methods and what performance can be achieved?

1.3. Thesis Statement

Main parts of this thesis were published on international Workshops and Conferences and are peer reviewed:

- Altinger *et al.* [13] commits to answer **RQ1** by performing a representative survey upon tools and methods.
- Altinger *et al.* [22][2]presents further insights into software methods and development procedures within the automotive industry. This work contributes to answer **RQ1**.
- Altinger *et al.* [23] releases an industry grade dataset containing software metrics on automotive software projects aiming to answer **RQ3** and **RQ3.1** by presenting correlation analysis upon those measurements.
- Altinger *et al.* [24] presents work on Software Fault Prediction (SFP) and CPFP answering **RQ2** by using machine learning classifiers to predict failures. Comparing the achieved performance values with literature **RQ2.1** will be answered. Correlation analysis and information ranking will be used to address **RQ2.2**. Main work will be on **RQ2.4** using state of the art literature methods and comparing their performance. Finally **RQ4** will be answered using a Principle Component Analysis (PCA) on the metric data.
- Altinger *et al.* [25] reports on influences of resampling algorithm to bug prediction performance. This work commits to answer **RQ2.3**.
- Altinger *et al.* [26] presents work on bug analysis to response on **RQ3.2**.

A detailed annotated publication list is given in Section A.1.

[2] This publication is submitted to review and is not published at date of release of this thesis

1.4. Thesis Organization

The thesis will be organized as follows. Starting with Chapter 2 to present the automotive industry as the research area and Chapter 3 containing the related literature with a focus on the field of fault prediction. Chapter 4 reviews a conducted questionnaire survey on tools used to specify, develop and test automotive software. An analysis upon three real world software projects is presented in Chapter 5. The obtained results on fault prediction are contained within Chapter 6. Finally Chapter 7 gives concluding remarks and a preview to further research topics.

2. Field of Study - Automotive Software Development

This chapter gives a short introduction to the automotive industry 2.1 and clarifies some domain specific environment parameters 2.2 along with common testing approaches 2.3. This examination will focus mainly on a computer science perspective.
For a more wider introduction to automotive engineering, including other disciplines such as computer science, the reader is redirected to Winner *et al.* [27], Braess *et al.* [28] and Crolla [29].

2.1. Automotive domain

Compared to the consumer electronic industry the automotive domain has a **rather long PLC**. Volpato and Stoccchetti [30] analysed cars PLC data between 1970 and 2006. They report on small cars to be redeemed by the new model after five years, premium cars after eight years, with a strong trend to shorter cycles. This is in line with Broy *et al.* [31] where they state a PLC is roughly seven to eight years, service and spare parts may last up to 15 years. According to the Kraftfahr Bundesamt (English: German Federal Motor Transport Authority) (KBA) [32] statistically cars in Germany are decommissioned after 8,8 years in use. Considering the average three to four year development phase as reported by Crolla [29], see Figure 2.6, some components development might be 18 to 20 years ago when a car is still on the road. Sabadka [33] predicts a reduction of a cars development time from 40 months to 25 in 2013 and further to 20 months in 2018. Using the VolksWagen (VW) Golf as a case story he analysis a PLC reduction from ten years in the late 1970is to three years in late 2000. In contrast, typical consumer products are replaced every two to three years according to Andrae and Andersen [34], software might be updated within much shorter cycles.

Most software runs on ECU with strict hard real-time constraints, memory and computing power is always limited. A modern day car can be seen as a heterogeneous network of up to 90ECU performing local and distributed computing tasks. Some nodes acquire data via a sensor interface, some pre-process data and some aggregate data, others control actuators. The automotive environment is rather harsh, *example given* the operational temperature is specified between -40° and +120°, shock, Electrostatic Discharge (ESD), vibration, *etc.* Tils [35] presented a rather good overview to all physical requirements to car electronics. These limitations may cause the Central Processing Unit (CPU) to run in throttled computation mode to fulfil operation requirements.

Hartung *et al.* [36] addresses the **variation diversity** within automobiles and visualizes them with examples. Pretschner *et al.* [2] uses 80 components which a customer can order, availability may depend on the country, to calculate 2^{80} variants an OEM can assemble electronics. During production the car is equipped with the ECU, but the actual software configuration is generated and deployed in the production line depending on the configuration the customer ordered. This causes a high number of conditions within the software, to cover all options. Peleska *et al.* [37] released an original software model visualizing the high amount of states and conditions to realize a simple car's turn indicator.

The Ultimate time goal: Start Of Production (SOP), the first day when a new model is built. This day requires all developments to be completed, all software to be tested and all certificates and accreditation documents to be issued. Long planning cycles are invested to solve logistic topics, all components need to pass qualification audits. Crolla [29] gives a brief overview to these milestones, see Figure 2.6. Once the SOP day is defined, customers may no longer order the old model, and logistic does not stock up components from the old model. This means, that from a certain point in time, it is not possible to extend the production of the old model anymore. Assuming a cycle time during assembly of about 70 - 90seconds and an average product price in the five digits regions, a production stop for a day can easily sum up to millions of EUR.

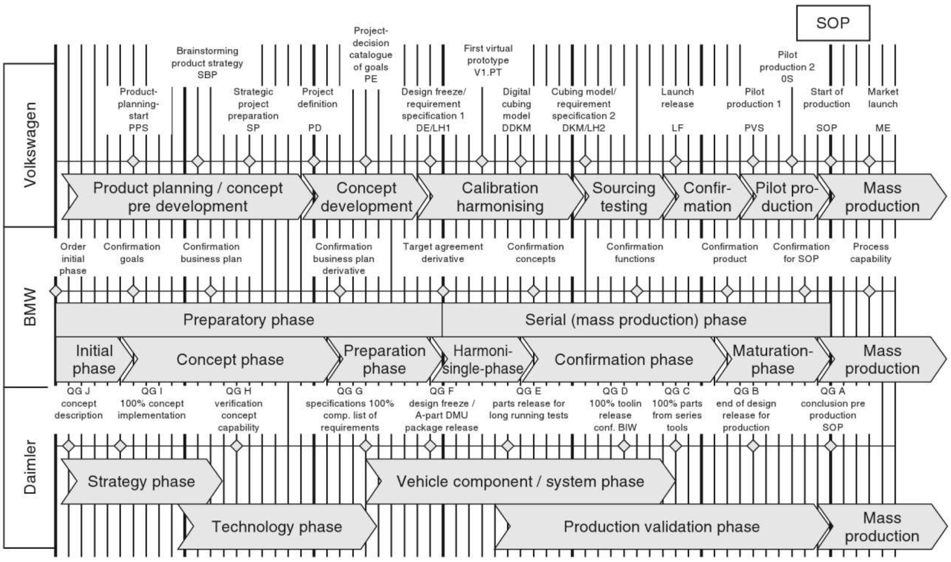

SOP
Volkswagen
Brainstorming product strategy SBP
Project-decision catalogue of goals PE
First virtual prototype V1.PT
Pilot production 2 0S
Product-planning-start PPS
Strategic project preparation SP
Project definition PD
Design freeze/ requirement specification 1 DE/LH1
Digital cubing model DDKM
Cubing model/ requirement specification 2 DKM/LH2
Launch release LF
Pilot production 1 PVS
Start of production SOP
Market launch ME
Product planning / concept pre development
Concept development
Calibration harmonising
Sourcing testing
Confir-mation
Pilot pro-duction
Mass production
BMW
Order initial phase
Confirmation goals
Confirmation business plan
Confirmation business plan derivative
Target agreement derivative
Confirmation concepts
Confirmation functions
Confirmation product
Confirmation for SOP
Process capability
Preparatory phase
Serial (mass production) phase
Initial phase
Concept phase
Preparation phase
Harmoni-single-phase
Confirmation phase
Maturation-phase
Mass production
Daimler
QG J concept description
QG I 100% concept implementation
QG H verification concept capability
QG G specifications 100% comp. list of requirements
QG F design freeze / A-part DMU package release
QG E parts release for long running tests
QG D 100% toolin release conf. BIW
QG C 100% parts from series tools
QG B end of design release for production
QG A conclusion pre production SOP
Strategy phase
Vehicle component / system phase
Technology phase
Production validation phase
Mass production

Figure 2.1.: The automotive milestone plan with SOP as the ultimate goal. Graphic is a licensed copy from Crolla [29] copyright granted by Wiley.

2.2. Development process

Automotive engineering uses the V-Model within all disciplines (powertrain, chassis, software, electronics, *etc.*). A recent survey by Bock *et al.* [38] reveals 100% of the interviewee automotive developers are familiar with this development approach. Schäuffele and Zurawka [39] explain this application and its adaptations to automotive in detail. In recent times the W-Development process, see Figure 2.2, is becoming more popular, as stated in a survey by Haberl [40]. This is an explicit testing oriented extension to the well known V-Process. Every specification stage has a corresponding testing stage. Bock *et al.* [38] conducted a survey among fifteen automotive software developers concerning their daily tools and methods usage. He present Matlab, Matlab/Simulink and TargetLink as the most dominating tools out of eight commonly known development products. All respondents are familiar with the V-development process, three quads of them with AUTomotive Open System ARchitecture (AUTOSAR).

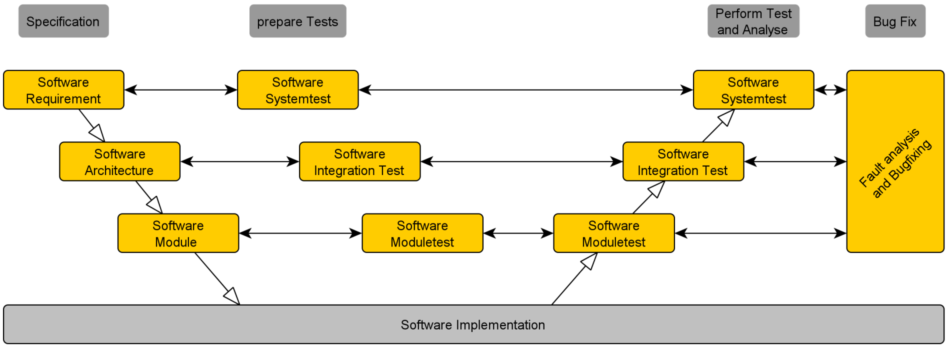


Figure 2.2.: The W-Development process as an testing oriented enhancement from the V-Model, originally presented by Jin-Hua *et al.* [21].

Following Broy *et al.* [31] and Bock *et al.* [38] **the majority of automotive software is developed using model driven or graphical programming approaches**, *example given* Matlab Simulink [41], with automatic code generation, *example given* using TargetLink [42]. Acting according to this process correlating code and model files are available during the development stages.

In accordance with the Motor Industry Software Reliability Association (MISRA) Software development guidelines [43], [44] there are various documents available during all stages:

- *specification:* functionality, timing, memory, processing time, *etc.*
- *software architecture:* modules, target ECU, schedule, *etc.*
- *interface description:* network message layout, method signature, *etc.*
- *static code analysis reports:* MISRA compliance, coverage, *etc.*
- *code review reports:* comments and suggestions
- *test reports:* TC pass rate, code coverage, *etc.*
- . . .

These documents are quite similar to other software engineering disciplines. A concrete workflow for software development is presented in Section 4.2.

A major difference is a rather strict development schedule containing the following milestones:

- *interface freeze*
  Where all network messages and interface (software and hardware) definitions have to be finalized. Beyond this point there is no change of communication messages or data-types.
- *software freeze*
  Where all software modules have to be finished and able to be called upon. There is no need for full functional implementation. Beyond this point no method signature changes are allowed.
- *100% software*
  Where all software modules have to be implemented and be able to pass functional tests. Succeeding this milestone only bug fixes are allowed to be submitted.
- *SOP*
  Where all software has to be finally tested.

Boogerd and Moonen [45] analysed a non automotive software from NXP Semiconductors discovering a similar behaviour. Within the early commits the bug rate increases where the authors conclude this is common behaviour to implement all features in the first place and later on to fix bugs.

As outlined by Schäuffele and Zurawka [39] **not all modules are developed by the same company**. Some do develop modules or components which they deliver as linkable binary. As stated by Pretschner *et al.* [2] an OEM may not even own a full Whitebox specification for third party modules. Another company might be responsible to integrate various modules together with an Operationg System (OS) to be executed at the target ECU. The AUTOSAR standard hosts abstraction layers and defines interface descriptions for all modules to communicate or use services provided by the OS to interact with the ECUs I/O. Figure 2.3 shows a simplified architectural layout. Several vendors offer AUTOSAR OS and basic software components via configurable code generators. The ECU manufacturer needs to implement the hardware drivers, whereas the OEM or system vendor only develops the application modules. As analysed by Dersten *et al.* [46] introducing the AUTOSAR standard to automotive has been beneficial to all software developing parties due to reduced costs on implementation and reuse capabilities. In addition standardized interfaces and Run Time

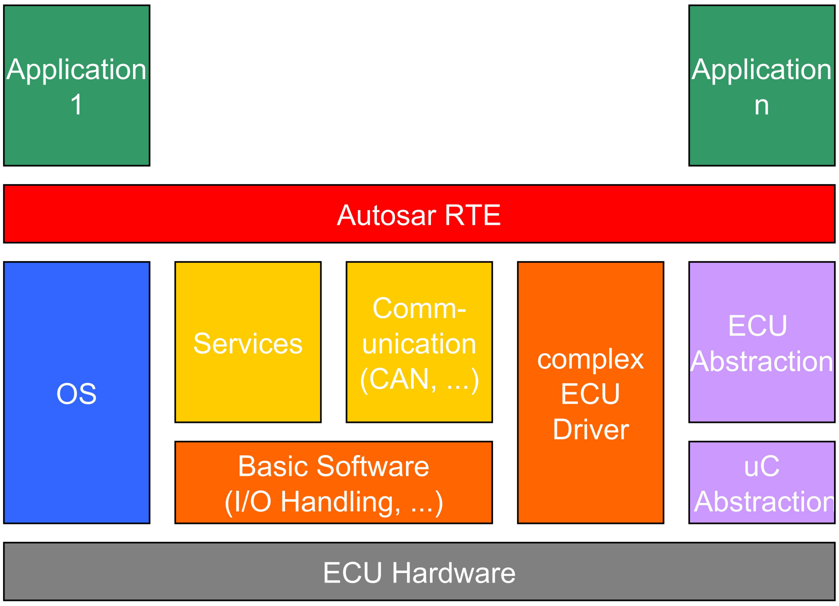


Figure 2.3.: The AUTOSAR architecture schematic as described by [47].

Environment (RTE) enabled developers to simulation and verify their application. The lower layer realizes access to hardware components and provides basic functionality *example given* logging and diagnosis services. Figure 2.3 visualises the architecture. The AUTOSAR RTE provides interfaces similar to an Application Programming Interface (API) and handles the data-flow. The application layers hosts the functionality, it might contain decision logics, controller software, *etc.*

As outlined by Schäuffele and Zurawka [39] the software is organized in modules partially hosting the functionality. A module can consist of a single or multiple software model. An application will assemble all modules and provide the external interfaces.

Developing a car in general requires to follow guidelines and requirements defined by laws (worldwide, national and maybe regional), in addition there are several Norms to be considered. Figure 2.4 lists the most common Norms. ISO/IEC 15504-2: 2003 (Automotive Software Process Improvement and Capability Determination (SPICE)) [48], delivers process documents, ISO 26262, ”functional safety”, [49], guides analysis of hardware and software and is a core part of every architectural decision. Projects are grouped into Automotive Safety Integrity Level (ASIL) QM and A to D, representing the severity of a failure’s consequence, see equation 2.1. *Severity* will range between ”no injuries” and ”life threatening” caused by fatal injuries, *Exposure* ranges from ”incredibly unlikely” to ”high probability” when cases occur under normal operations, *Controllability* ranges between ”controllable” to ”difficult or unable to control”. A system classified as ASIL D will cause harm to human life if it fails in which a highly likelihood of situations occur. A visual example of such a system might be a highly automated car leaving the road due to a software flaw and injuring its passengers in the case of a crash. An ASIL QM functionality will cause no harm in case of failure *example given* a satellite navigation system.

$$Risk = Severity \cdot (Exposure \cdot Likelihood) \tag{2.1a}$$

$$ASIL = Severity \cdot (Exposure \cdot Controllability) \tag{2.1b}$$

2.3. Testing process

Testing a car is a complex task guided by multiple regulations and defined by detailed processes. Following the W-Model, see Figure 2.2, various testing stages have been applied in recent times. Parallel to writing functional requirements dedicated test engineers write test specifications. Every module has to pass multiple tests on test benches prior to its integration assembly in a prototype car.

These are most common the **”in the loop”** tests, *example given* Model in the Loop (MiL), Software in the Loop (SiL), Hardware in the Loop (HiL). These test beds realize module tests in various integration levels. Using model driven approaches *example given* a Simulink Model is the first deliverable to be tested using MiL to pass functional tests for sub functions organized within a single model. These model files are used to derive a generated code which is put on a SiL test, mainly repeating functional tests to ensure correct code generator settings. Putting all modules together and running integration tests is the second part. A SiL will be executed as a simulation running on a computer. Within this stage the *interface freeze* milestone

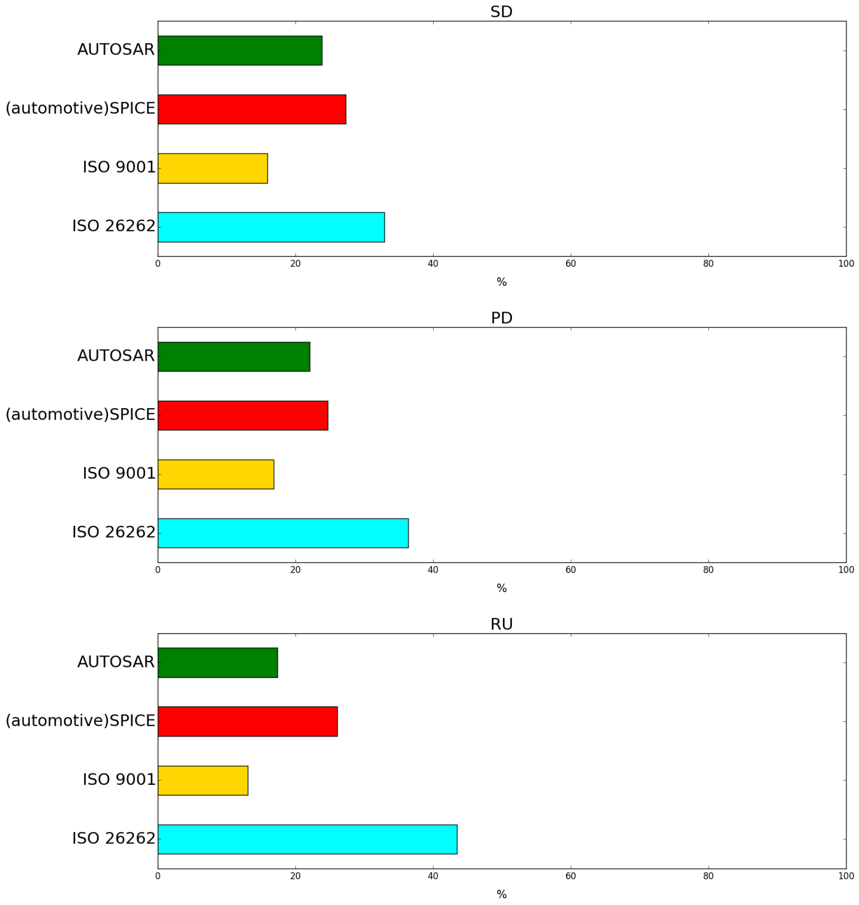


Figure 2.4.: Technical Norms to be known and considered by developers departments as a result of a Survey by Altinger *et al.* [13].

could well be accomplished. MiL and SiL tests are used to test algorithmic implementations. Various test stimuli will be generated based on white box testing methods to directly target an implementation. Processor in the Loop (PiL) is the first integration together with the OS running on the target CPU which is a component of the ECU. If the ECU is not available at this time, the tests are executed on an evaluation board or skipped. All stimuli are realized as direct function calls from a test environment as the external communication (*example given* CAN) might not be available at this point of integration. HiL tests are executed on the test bench containing multiple ECU and communication simulators, often actuators are present too. See Figure 2.5 for a representative HiL test bench at Audi AG. The software needs to run on the target ECU, the test stimuli is generated via communication messages, *example given* a CAN message or an electric signal simulating a switch or a sensor value. These communication messages might be automatically extracted from the message databases for CAN and FlexRay where scripts can extract desired messages for each specific ECU as the System Under Test (SUT). As Bergmann [50] explains, classic HiL were extended to ”connected HiL” where the cars whole electronic environment is installed and tested. The system needs to pass real driving scenarios where even sensor inputs are simulated as presented by Müller *et al.* [51] and Pfeffer *et al.* [52]. Within this stage the *100% software* milestone could be accomplished. HiL test's stimuli will focus on realistic interaction, this means a TC might look like: ”turn on indicator and wait for 30 seconds”. Figure 4.13 visualises tools and input data related to this ”in the loop” tests. The final testing stage is a prototype car, where the ECU is integrated along with the target sensors and input devices. Another task at this stage is to test power consumption and sleep respective wake up time. The car is required to pass a number of user interaction tests and perform a defined mileage per driving scenario as required by the law. **If a bug is discovered all prior test cases and stages have to be executed again**, which is one of the reasons for higher testing costs if a bug is discovered late.

A comprehensive overview to development and testing activities on an ADAS will be given by Müller *et al.* [51]. The authors describe a distributed system and the challenges on designing a test application to qualify the system. The authors explain in detail testing approaches starting at the environment simulation and generate input data to be fed into real sensors

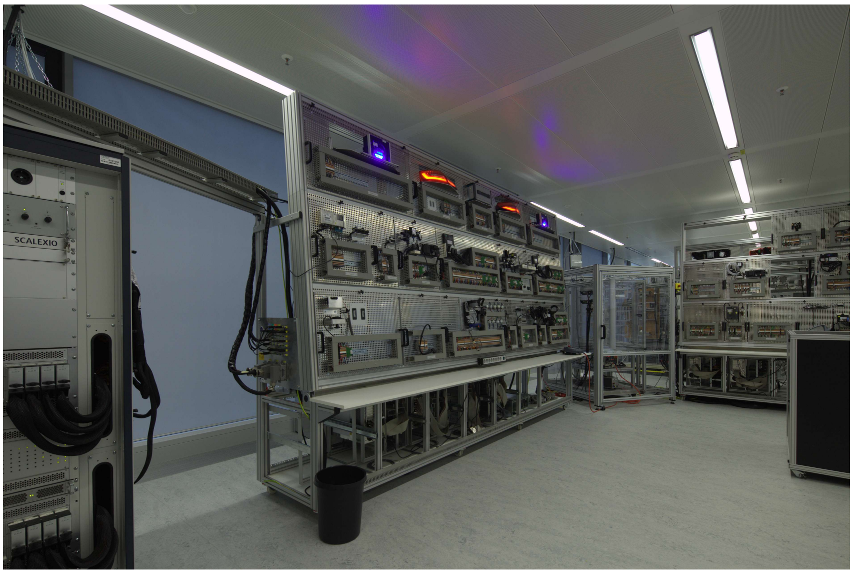


Figure 2.5.: Exemplar picture showing a classic HiL test bench, copyright AUDI AG

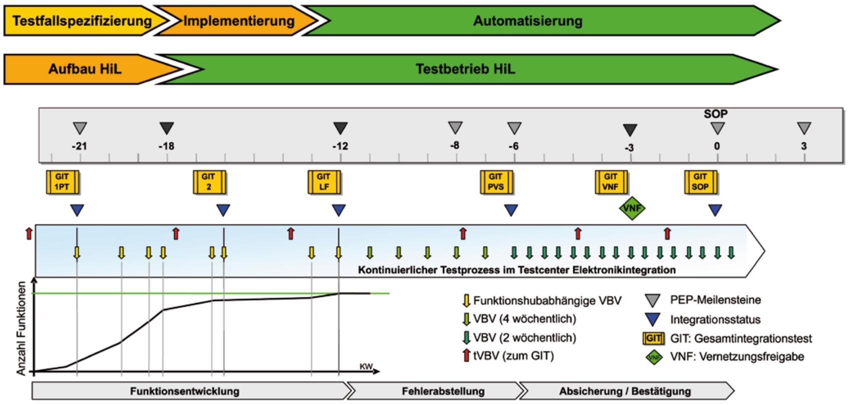


Figure 2.6.: Exemplary project plan containing milestones when developing an ADAS. The graphic is a licensed copy from Müller *et al.* [51] copyright granted by ATZextra/Springer.

mounted at a HiL test-bed. They use video screens to visualize the scenery which can be captured by the car's camera. Further they generate ultrasonic sound according to simulated distances for every ultrasonic sensor. These sound waves are captured by the original sensors and processed by the target ECU. Figure 2.6 is a licensed copy from this publication. It shows the milestones required to set up the HiL and test the ADAS starting 23 months prior SOP. The required time to implement the ACC feature is estimated to 12 months, which is the same time required to perform all defined TC.

3. Review of Related Work

This chapter provides an overview of literature related to Software Fault Prediction (SFP) on public available software metric repositories. A complete chronological overview about this chapters work can be extracted from Table 3.1. The following subsections cover dedicated research topics. Most of the publications cover a multitude of these topics, thus there is no complete review of a single work. A majority of the papers will be partly reviewed within multiple subsections. Each section will present the reviews in chronological order. This chapter contains primary literature sources but also reports about summaries stated by three systematic literature reviews.

The first part 3.2 introduces available software metrics and their limitations, the second part 3.3 presents reports concerning fault prediction, the third part 3.4 lists datasets capable to perform SFP with public access, the fourth part 6.1 presents work on SFP and reviews different methods from the last 30 years. The Fifth part 6.3 consists of reviews on a recent field of study, CPFP, whereas the sixth part presents the imbalanced class distribution. The final part 3.8 will be an overview to prior analysis on error distributions within software projects.

3.1. Empirical Evidence upon Automotive Testing Methods and Tools

Zhang and Pham [68] reports upon 32 factors to influence software reliability. The authors conducted a survey with 22 responses. Their data is based on a survey with 22 responses among them one automotive company. Their analysis states testing coverage, testing effort and testing environment do have significant influence, testing tools are centrally ranked. According to the data testing is correlated with requirements and working standards

Table 3.1.: A timeline representing the reviewed literature between 1996 and 2016

1976	Mccabe [53]
1977	Halstead [54]
1979	Curtis *et al.* [55]
1981	Basili and Phillips [56]
1984	Adams [57]
1985	Sherif *et al.* [58]
1987	Boehm [59]
1994	Chidamber and Kemerer [60], Fenton [61]
1996	Khoshgoftaar *et al.* [62]
1997	Khoshgoftaar *et al.* [63], Pfleeger *et al.* [64]
1999	Fenton and Neil [65]
2000	Fenton and Ohlsson [66], Graves *et al.* [67]
	Zhang and Pham [68], Japkowicz [69]
2001	Boehm and Basili [70]
2002	Briand *et al.* [71], Denaro and Pezze [72], Ostrand *et al.* [73]
2003	Menzies *et al.* [74], Drummond *et al.* [75]
2004	Guo *et al.* [76]
2005	Do *et al.* [77], Ostrand *et al.* [78], Sayyad and Menzies [79]
	Vipindeep and Jalote [80], Wagner *et al.* [81]
2006	Bell *et al.* [82], Broy [7], Kim *et al.* [83]
	Li *et al.* [84], Nagappan *et al.* [85], Tomaszewski and Damm [86]
2007	Broy *et al.* [31], Krisp *et al.* [87], Menzies *et al.* [88]
	Mizuno and Kikuno [89], Ostrand *et al.* [90], Pretschner *et al.* [2]
	Weyuker *et al.* [91], Zhang and Baddoo [92], Zimmermann *et al.* [93]
2008	Boogerd and Moonen [45], Gondra [94], Jiang *et al.* [95]
	Kamei *et al.* [96], Lessmann *et al.* [97], Lincke *et al.* [98]
	Moser *et al.* [99], Vandecruys *et al.* [100]
2009	Boogerd and Moonen [101], Catal and Diri [102], Catal and Diri [103]
	Hall *et al.* [104], Herraiz *et al.* [105], Mende and Koschke [106]
	Mockus [107], Schneidewind [108], Singh *et al.* [109]
	Turhan *et al.* [110], Zimmermann *et al.* [111]
2010	Causevic *et al.* [112], Dambros *et al.* [113], Menzies *et al.* [114]
	Mizuno and Hata [115], Khoshgoftaar *et al.* [116]
2011	Haberl *et al.* [40], Posnett *et al.* [117]
2012	Dambros *et al.* [118], He *et al.* [119], Rahman *et al.* [120]
	Rommel and Girard [121]
2013	Herbold [122], Mathworks [123], Nam *et al.* [124]
	Radjenovic *et al.* [125], Weiss [126]
2014	Abaei and Selamat [127], Jus *et al.* [128], Zhang *et al.* [129]
2015	He *et al.* [130], Menzies *et al.* [131]
2016	Bock *et al.* [38], Klaus [132]

(norms, coding standard, *etc.*). A single programmer's skills or the target hardware is not reported to correlate to testing effort.

Several white-papers exist, *example given* Krisp [87] or Mathworks [123], reporting about one tool with its successful application. Within the majority of these publications the tool vendor and the publisher are from the same company or the author is closely related and the scope is very limited. Market research reports like [121] can give quantitative data but most of the time they do not report the number of respondents or contacted persons or present an outlook to future trends.

Haberl *et al.* [40] reports about an annual survey conducted within the Austrian, Swiss and German software industry with no specific focus on the automotive sector. During the 2011 edition they collected 1.623 responses where they asked all participants to answer 100 test related questions. The data presentation is divided into three groups: managers, developers and testers. More than 40% of the participants are either software testers or test managers. More than a third of the projects uses the V, 14% the W, up to 10% the waterfall model. The majority of small[1] companies applies agile, mainly Scrum, projects. Large enterprises[2] may use agile methods in up to 19% of their projects. Roughly half of the participants report of some defects discovered after the software has been released, only 3% list severe defects report to the customer, thus the authors claim software quality and testing efforts do work as up to 65,7% report upon dedicated testing and quality engineers and only 2-5% software developers do act as quality engineers too. More than 80% of the TC are clearly designed with up to 50% formulated in free text or oral form even containing in up to 72% of cases pre calculated response values. TC are execute in dedicated testing environments in up to 84,1% of the cases, however, 24,1% still use the real live system. Three dominating TC performance metrics are reported with *requirements coverage* at 75%, *TC execution rate* by 60% and *code coverage* in 25% of the cases where in 77,8% the test activities are finished when each requirement has been checked at least once. More than 80% of the TC are executed as regression tests. The authors report of a high usage of test automation and test exe-

[1] up to 100 employees

[2] more than 1.000 employees

cution tools, but do not list any. Static code analysis is used in up to one third of the projects, where more than 50% perform tool supported coding standard checks. Tests following the black box methodology are dominant in 92% of the tests, white box at 82%. Compared to the authors survey in 1997 testing gained more importance and is performed in a more systematic way (dedicated testers, pre designed TC, certified testing processes).

Another industrial survey is reported by [112] covering 83 respondents origin in the IT industry with no further limitations. The authors categorize three different views: job description (tester/none tester), type of application (safety critical/none critical) and development target (desktop/web/embedded development target). Upon their analysis a common team size is between one and ten engineers. They present unweighed free text answers by some testers regarding their tool usage. The majority uses tools to execute regression tests or an Integrated Development Environment (IDE) to perform debugging and manual testing. The majority of tools listed is open source and dedicated to run on a computer. They presents a list of dissatisfying topics among the developers: Changing requirements during coding phase is top listed, begin of coding phase before finishing design and no comprehensive documentation are high ranked as negative influences. Surprisingly test driven development is no in use in practice, but participants wish it to be practice.

3.2. Software Metrics

Every engineering domain requires objective measurement to rate and evaluate processes and systems. In terms of computer science this refers to the topic of software metrics. This section introduces their origins and tools to measure them. Originally used as indicators for quality aspects and project monitoring software metrics are among the oldest research area connected to fault prediction.

Curtis *et al.* [55] performed tests inviting 54 professional developers with six years of experience on average to analyse Fortran code snippets. The participants where requested to find bugs within the presented code. The study used different code styles and analyses their influence upon time required to find bugs. The participants were guided with software metrics which they could use as an indicator. Upon the studies results LOC, Cyclomatic Complexity by McCabe (CC)[1] and Halstead Effort (HE)[2] perform similar on small subroutines. If the code is longer the power of LOC decreases where HE is the best indicator. The authors analysed Halstead effort to perform better as a prediction for psychological complexity.

Boehm [59] released a list of common development issues within industrial software and their representative metrics. The author reports only of his experience as a test engineer and presents no evidence, but this remains one of the very few publications of its kind.

Radjenović *et al.* [125] presented an overview to metrics and their success on SFP. LOC is powerful at the pre release stage, CC[1] might be good for big projects and becoming strong in the post release phase. Halstead metrics[3] might be good at pre release state, but overall weak. Strong predictors performing well within pre & post release and small & big code bases are code chrun, file and change history. Their overall analysis on the usage of metrics showed the majority of publications uses Object Orientated (OO) metric like Chidamber Kemerer [60] OO metric (CK) by 49%, classic source code metrics like LOC & CC by 27%, but the smallest occurrence by 24% is to process metrics like code chrun.

This review focuses mainly on metrics suitable for function orientated programming languages as the analysed software, see Section 5.1, has been written in C. Therefore OO metric suites, *example given* the CK metric suite defined by Chidamber and Kemerer [60], or software reliability & quality metrics are not considered. A full introduction onto this topic is given by Schneidewind [108].

[1] see equation 3.1
[2] see equation 3.2f
[3] see equation 3.2d - 3.2f

3.2.1. Presentation on selected Software Metrics

LOC is the easiest and oldest software metric which exists in various forms: total LOC, LOC with or without comments, number of comment lines, number of executable statements, number of variables, *etc.* Within most of the reviewed literature LOC is used in total or as statement count as all other forms (number of statements, number of comments, *etc.*) are highly correlated.

McCabe *et al.* [53] present their graph theoretic complexity measure, CC. A core part is to be independent from the program's size, which means adding or removing a simple statement does not affect the measurement value. The CC is calculated according to equation 3.1.

$$v = e - n + 2p \tag{3.1}$$

Where v is CC, e is the number of edges, n the number of vertices[3] and p the number of connected components[4]. The authors list typical programming elements and their CC: sequence: 1, if then else: 2, while-loop: 2, Main program with two subroutines: 6, see Figure3.1 for illustration. Overall CC can be interpreted as the mental effort a developer has to invest to understand all possible decision paths within a program.

Halstead *et al.* [54] claim a similarity to physics stating a software algorithm containing characteristics which are measurable, thus they present complex metrics based on the number of operators[5] and operands[6].

- η_1: the number of unique operators
- η_2: the number of unique operands
- N_1: the total number of operators
- N_2: the total number of operands

With these acquired values the performance figures can be calculated:

[3] the number of states within a state diagram

[4] *example given* a subroutine called from a main program counts as $p = 2$

[5] *example given* +, -, –¿, printf(), method calls, *etc.*

[6] *example given* variables, static numbers, format instructions for printf, *etc.*

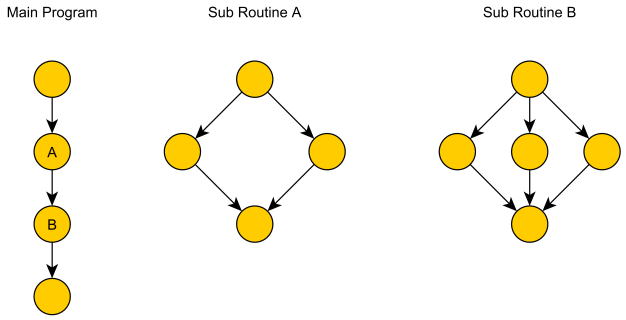


Figure 3.1.: example to CC: A main program as a sequence with two sub routines as if then else branches

Program vocabulary:

$$\eta = \eta_1 + \eta_2 \tag{3.2a}$$

Program length

$$N = N_1 + N_2 \tag{3.2b}$$

Program length calculated

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \tag{3.2c}$$

Program Volume

$$V = N \cdot log_2 \eta \tag{3.2d}$$

Program Difficulty

$$D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2} \tag{3.2e}$$

Program Effort

$$E = D \cdot V \tag{3.2f}$$

Basili *et al.* [56] has been among the first to publish work on software metrics and development effort. They analysed a NASA ground support software for satellites written in Fortran. HE and CC are compared against weekly time table reports collected from the developers. Among their Pearson correlation analysis Halstead effort is highly (0,6774) correlated to the time required to write the code. In addition they present a strong (0,654) correlation between Halstead effort & CC and an even stronger correlation between executable statements with Halstead (0,8301)& CC (0,9116). Overall the correlation between the single metrics and the number of errors is between 0,4861 and 0,5837, which is a weak correlation. Using a combination of their metrics they could archive a correlation against the error count up to 0,6227.
A full list, as available by 1985, of software metrics and their interpretation is given by Sherif *et al.* [58]. The authors present an extensive literature review upon software metrics to measure programs distributed along the full software live cycle.
Fenton *et al.* [61] evaluated software metrics upon their measurement fundamentals. They request to obey measurement theory principles when measuring software as in is common to all other scientific domains. Those principles are listed as the request of a measurement model, a scale and the ability to represent an absolute order of measurement data. The authors claim that software complexity may not be expressed in a single $\in \mathbb{R}$ number with ordinal order as there is no general expression possible for Software complexity. They advise to use multiple internal attributes to derive measurement, a single attribute is misleading. The authors stress that if a developer adds one LOC the complexity can not decrease, which needs to be considered in the metrics. Upon their analysis CC and HE satisfy this requirement.Pfleeger *et al.* [64] releases a report on the usage of software metrics among practitioners. Within their analysis they state that developers might use whatever metric they have a standard spreadsheet for, even if the method is not correct.

3.2.2. Performance

Zhang and Baddoo [92] report on work on performance comparison between HE and CC metrics using only bug fix updated on the Eclipse JDT

Open Source repository. The authors perform Pearson correlation analysis between the metric values and the number of bugs. In general HE and CC are both positively correlated to the number of bugs, CC might be stronger than HE. Eclipse follows the OO development approach, containing a high amount of small statements due to the encapsulation and inheritance features. This implies that a single component may contain a lower number of operands and operators. Which may cause HE to perform lower than in function orientated programs. In addition they list a correlation between LOC, CC and HE against the number of bugs. Within their findings neither complexity nor code volume (LOC) are the single source of bugs. An in-depth analysis of single files supports their thesis on more complex files not necessarily containing more bugs. Summarizing, they state HE performs better in larger software than CC or LOC, yet all three metrics types are strongly correlated to each other. Their final remark: ”Therefore using McCabe’s cyclomatic complexity metric and Halstead’s effort metric should be a good combination of metrics for capturing the complexity of a software system.”

Jiang *et al.* [95] analysed the influence between two categories of metrics, code metrics (LOC, Halstead E,V,D, *etc.*) and design metrics (edge count, node count, CC, branch count, *etc.*) seeking their capability to predict failures. The authors based their evaluation upon the NASA metric data program [133] (NASAMDP) where they selected 13 projects written in C,C++,Java and Perl. All code metrics were originally supplied by the dataset, design metrics needed to be reverse engineered using the source code. According to the authors’ comments, this is a common approach. Within their experiment they created three groups for each dataset: code metric only, design metric only and all available metric. As the prediction model they selected five machine learning algorithm from the WEKA toolkit [104]: Random Forest, Bagging, Logistic regression, Boosting, and Näive Bayes (NB). The results have been evaluated using a ten fold cross validation. According to their analysis no single machine learning algorithm performs best among all datasets and groups, Random Forest (RF) among the top for 9 out of 13 datasets. Using design and code metrics performs overall best, whereas the code metrics only group performs in 7 out of 13 cases identical to the all metrics group. Summarizing their work they state code metrics outperform design metrics, but using all available metrics might be the best. Based on the performance data,choosing the correct set of metrics (and the program to calculate them) is more important than the decision upon the model building (aka machine learner) algorithm.

Moser *et al.* [99] evaluate if code metrics archive the same fault prediction accuracy than change metrics. Their datasource is based on the Eclipse bug dataset provided by Zimmerman *et al.* [93]. Their analysis is designed very similar to Jiang *et al.* [95], but they are using change metrics (code chrun, file and bug history, *etc.*) instead of design metrics. The performance reports a code metric only prediction is outperformed by a change metric only prediction which are outperformed by a combination of both. This is a consistent finding upon Menzies *et al.* [88] and Jiang *et al.* [95]. All three publications are using machine learning approaches.

Posnett *et al.* [117] investigated the influence of different aggregation levels. In terms of software metrics and their application to SFP this refers to measuring at the file or the package level. The authors gather data from eighteen Apache projects written in Java, measuring LOC, number of authors per file, number of active[7] authors, number of improvements, number of new features, code chrun and the file history. All metrics have been acquisitioned at file and at component[8] level. They build up a Logistic Regression (LR) based predictor and use Area Under Curve (AUC) as the predictive measurement. Overall the file level based metrics report better AUC values. Within their findings multiple metrics show no significance at the package level, but a high one at the file level (*example given* the number of active developers). Their analysis unveils that looking at module level is too coarser and might remove precision in terms of AUC when performing SFP. Some metric show even different tendency at package and file level.

3.2.3. Tools to Collect Metricdata

There exists only one report evaluating different metric gathering tools by Lincke *et al.* [98] evaluated ten tools to derive OO software metrics, but

[7] an active author did commit code changes during the actual commit

[8] a component might consist of multiple files, one file might contain multiple methods or classes

include volume metrics (LOC, *etc.*). They report on a case study using 100 random Java programs between 5 and 500 source code files with an active user base and a high ranking, all programs are hosted on SourceForge.net. They presented obvious measurement differences, some measured metrics differ between 6% and 80% among the various analysed tools. The concrete numerical values strongly depend on tool implementation and may not be comparable. Their analysis does not cover Halstead metrics, but rather suggests a similar behaviour.

3.2.4. Summary

There exists a wide range of software metrics, most of them correlated with LOC. Work by Moser *et al.* [99] and Jiang *et al.* [95] empirically demonstrates the power of selecting multiple types of metrics when using machine learning predictors. This correlates with the suggestions made by Menzies *et al.* [88], where the author states using all available data when applying machine learning algorithms. Thus the selection of metrics is more crucial than the choice of fault prediction algorithm as demonstrated by Jiang *et al.* [95]. Weyuker *et al.* [91] state change metrics (file and fault history) are strong fault indicators if the software is more mature. This statement is supported by D'Ambros *et al.* [113] and He *et al.* [130]. Following Lincke *et al.* [98] the tool selection is important as there are significant numerical differences between various tool implementations. Posnett *et al.* [117] suggest measuring metrics at the file level due to better predictive performance. Fenton and Neil [65] discuss differences between academic metric selection (complex, multiple metrics, mathematical valid, *etc.*) and industrial (LOC only, use whatever an available tool provides).

3.3. Case Studies

Within recent years there have been two types of SFP and CPFP case studies, one group using industrial code bases, see Section 3.3.1, the other group uses code metric datasets collected on open source software repositories, see Section 3.3.3.
Catal *et al.* [103] released a systematic literature review in the field of SFP.

They discovered an increasing number of publications after the 2005 release on the Promise dataset by NASA. This has been the first time that a dataset has been publicly available containing code metrics and bug data. Upon their analysis they identified more machine learning approaches after 2005. In general the research field is rather young, starting with the first publications in 1990 and finding its first peak after 2005. In their opinion it is not possible to transfer a statistical model specialized for one project or company for another project or company, which defines the field of CPFP.
Radjenović *et al.* [125] released a systematic literature survey reviewing 106 publications issued between 2001 and 2011. In their work they identify 105 different software metrics executed on 106 datasets. The authors complain that only a fifth of the studies uses publicly available data, all others might not be repeatable. Only half of the datasets contain an adequate (size, type of software, distribution, purpose, programming language, *etc.*) description. During their review they identified three types of metrics: traditional (LOC, CC, *etc.*) - used within 27% of the studies, process (file and fault history, code chrun, *etc.*.) - used within 30% of the studies, and OO (number of classes, *etc.*) used within 43% of the studies. All reviewed studies have been evaluated on hand written code.
Common to all industry case studies, the authors never released their metric data and most of them do not even name the analysed software products by names or used arbitrary indicators. Overall the reported prediction performance is better on industrial code than on open source software. Table 3.2 shows a summary of all used case stories presented within Section 3.3.1 and 3.3.3. The reviewed work has been selected due to multiple publications using the same dataset. An initial publication reports on prediction performance (*example given* Ostrand *et al.* [73]), a later one analyses fault distribution (*example given* Bell *et al.* [82]) or uses different prediction methods (*example given* Ostrand *et al.* [78]).

3.3.1. Industrial with a Private Dataset

First work on industrial fault prediction has been presented by **Khoshgoftaar *et al.* [62]** where they analysed a Telecom software with 1,3 Million LOC. The authors focused on code metrics as they were collected from standard quality metrics. The values have been normalized before used within their model. In their analysis the authors state that new modules tend to contain more faults than unchanged modules from the previous release. They used a discriminant analysis where they added variable (metric) to model starting with the highest significance to faults. Summarizing their work it is possible to predict faults, but every project needs to derive its own model. Using their derived models they could achieve a rather high performance (31,1% misclassification) on prediction.

Their work has been followed up by **Graves *et al.*** [67] using the same Telecom software as a case study. The state of the code has been changed roughly a 130.000 times performed by a few hundred developers within a two year period. On peak faulty modules contain 120 bugs per year. Their focus is on module[9] change history as other metrics are too strong related to LOC. They introduced new measurements: number of past faults, average age of code as representative share between new added code within the last commit & existing code and a weighted time damp symbolising the number and size of recent changes to the file. Within their analysis the number of developers is a minor bug indicator as well as CC and the number of simultaneous changed modules per commit. Their strongest indicator has been weighted time damp and the number of past faults. To predict faulty modules they used Generalized Linear Models (GLM) with a Poisson error distribution. The authors mainly present a rating representing the influence of various metrics to predict the number of faults, but never evaluated the precision of their models.

Ostrand *et al.* [73] performed SFP on Telecom software at AT&T Labs summing up to 0,5 Million LOC. They observed that highly faulty files may not stay faulty within the next release, which might be caused by a more intense testing on previously known buggy files. In their observations a few number of files contain a high percentage of faults. If the software is more mature (in terms of higher release number), the concentration may be even higher due to a more concentrated development on specific modules. Newly added files might contain more faults than older ones, which might be caused by a more intense testing at early commit stages. Larger modules contain a lower fault density than smaller modules.

This work was continued by **Ostrand *et al.*** [78] three years later. They present a use case study on two industrial applications from the Telecom

[9] a module contains multiple files

sector originally developed at AT&T Labs. Again they state that 20% of files contain 80 to 93% of faults. They used a Negative Binomial Regression (NBR) model to predict failures. Their aim has been to suggest in which code regions testing would be more beneficial than a strategy of how best to select TC. They used two types of metric, code metrics (Halested, $ln(LOC)$, CC) and change metrics (new or old file, the file has been changed within actual commit, number of commits since the initial as representative to file age, $\sqrt{fauls_{prev}}$ where $faults_{prev}$ being the number of faults within all previous commits). They discovered CC to be too similar to LOC. During their work they set up two predictors, one using the full set of metrics, the other having been reduced to LOC only. In their opinion the simple model delivers a reasonable performance with 70% of all detected faults compared to an average of 83% on the full model. During their analysis they discovered the full model performed better if the code is mature (which they define as containing less errors due to a higher number of commits). During their analysis they defined a commit affecting one to three files as a bugfix, if it affects many files it might be a feature enhancement or interface change or a code revert. Overall they discovered a significant influence to fault proneness by the developing language.

Their work has been followed up by **Bell *et al.* [82]** who also conducted a user case story on industrial Telecom code developed at AT&T. Four different projects have been analysed, written in various programming languages, however, not all of them have been released on a fixed schedule, leaving shipment whenever a module has been finished. They confirm LOC metrics perform well, but are outperformed by change metrics. They state that fault proneness decreases with the file's age, meaning the longer a file is within the repository, the less likely it is to contain faults. During their performance analysis they found fault history to be usable to predict faults, but less powerful than file change history. One of the most influential parameter is called file exposure, referring to the time a file has been used within the analysed software system. Along with Ostrand *et al.* [73] and [78] they found the LOC only (using $log($LOC$)$ and an indicator if the file is new) model has only got a 10% lower performance than the full model (LOC, file age, $\log(exposure)$, $\sqrt{number \quad of \quad changes \quad last \quad Month}$ and a coefficient for the programming language). They report on using binomial regression once again, however, they split their data into a training and testing set.

Nagappan *et al.* [85] performed a use case study with five object oriented products[10] developed by more than 250 engineers at Microsoft. Their focus has been on post release failures. Similar to findings by Bell *et al.* [82] they state that the past fault history can be used to predict future (post release) faults. They used linear and logistic regression to build up their models. Their main input has been on static metrics (number of classes, number of functions, *etc.*.). Their strongest fault indicators are the number of classes, the number of functions and the number of variables, CC and classes with a coupling to C methods. Overall they report on low prediction performance but scattering with the actual project. They state that software development methods have influence if the selected metrics correlate with failures. One of their analysed projects did monitor various metrics and kept them below a certain threshold. This leads to none of the metrics correlating with the failure rate which prohibit the usage of SFP. Further the authors made initial tests on CPFP stating that prediction models may be transferred to other projects if these are similar, but no single software metric suite exists to predict failure for all projects.

Zimmermann *et al.* [111] continued and extended the work of Nagappan *et al.* [85]. They analysed four Open and eight closed source projects hosted and developed at Microsoft. Their main focus has been on CPFP where they are one of the first to report performance data. Overall they achieved a low success rate. Only 3,4% of 644 cross project experiments successfully predicted failures using a model trained on another project. This sums up to a weak CPFP. They only considered a successful CPFP if the recall was above 0,75, precision and accuracy was good. They used typical static volume code metrics (LOC, CC, *etc.*), change metrics (defined as code chrun, measuring LOC added, LOC removed or the number of LOC changed) and process metric (number of developers, file age, *etc.*) similar to Ostrand *et al.* [73], [78], Bell *et al.* [82] and Nagappan *et al.* [85]. In contrast to those publications Zimmermann *et al.* [111] used logistic regression and normalized their metrics values using LOC. They name a better prediction performance when using normalized values. Within their analysis they state that CPFP is only possible if the project settings are similar. They list factors

[10] using C, C++, C#

influencing similarity, among them tool settings (*example given* compiler, editor, static code checkers, *etc.*), software development processes, target OS platform and code reviews. Their analysis shows that some projects tend to be more related than others. The authors define a similarity vector between two projects to a derived decision tree if CPFP is possible.

3.3.2. Industrial with a Public available Dataset

Within this section reports on SFP using a public available dataset are presented. Thus these experiments are repeatable. The only industrial available dataset has been released by NASAMDP and the PROMISE repository.
Menzies *et al.* [88] has been the first to use machine learning approaches to build up defect prediction models. To predict error prone software modules they use three different machine learning algorithms (OneR, J48, and NB) included in the WEKA [104] toolkit. To evaluate their prediction they used the NASAMDP and the PROMISE repository of software engineering data. Within their analysis statistical Methods (NB) outperform others leading to the statement that the choice of learning algorithms clearly is more important than the selection of metrics. They could find no clear deviance that single metrics outperform others or that LOC is the ultimate metric. Their suggestion is to use multiple metrics, at minimum three, at best all available. During their experiments they were limited to static metrics due to the used datasets. Their performance is high, as they report upon finding 71% of defects with a false positive rate of 25%, which is clearly better than other methods they list in comparison, *example given* manual code review discovers 60% of defects.
Using a similar technique **Gondra [94]** presents work on performance comparison between neuronal networks and other machine learning approaches. They analysed the sensitivity of software metrics on fault prediction using the NASAMDP. Within their results LOC performs best but CC is listed as the third strongest indicator. Comparing their performance reports Support Vector Machine (SVM) outperform Neural Network (NN) with 87,4% to 72,61% of true positives. They claim this is due to defect predictions nature as binary classification problem which is the SVM domain.
Vandecruys *et al.* [100] presents work on the NASAMDP. They present AntMiner+ a data mining tool based on the Ant Colony Optimization (ACO)

algorithm which is inspired by biological ant colonies. A path's weight is determined using the number of ants previously travelled. The more ants used a path the more likely other ants would follow this. The authors claim a novel application of ACO to fault prediction. As available by the dataset they use LOC, CC, Halstead (error, effort, *etc.*) and call statistics as their input metric. The main focus has been put on comparing ACO to C4.5, logistic regression and SVM using the WEKA toolkit [104]. Bare performance values suggest a lower detection rate by AntMiner+ compared to C4.5 but equally to SVM. AntMiner+ main benefit is to extract a small rule set for classification between faulty and bug free commits. Within the rules the authors list Halstead content and LOC metric as the most important ones.
Summary: SFP seems to work within industrial settings as early studies report successful application (Ostrand *et al.* [73], [78]) using statistic binomial regression, later (Menzies *et al.* [88]) using machine learning. Models based on LOC metrics seem to perform well, additional metrics (code chrun, developer, file and fault history) slightly enhance prediction performance. Using logarithmic or normalized metric values seems to be beneficial. Menzies claims to use all available data when applying machine learning algorithms. Some applications on CPFP exist (Nagappan *et al.* [85],Zimmermann *et al.* [111]) but report a low performance when transferring defect models. Both state that a similarity between projects is beneficial for CPFP. Transferring defect models between releases of the same software seems to perform well as reported by Zimmerman *et al.* [93].

3.3.3. Open Source

A series of SFP using open source software exists. In contrast to previously introduced industrial case studies 3.3.1 the source code is available for further inspection and reproduction upon the reported experiments.
Zimmerman *et al.* [93] present work on an Eclipse dataset covering releases 2.0, 2.1 and 3.0. They use linear regression to predict faults within one release and cross release. Initially they performed the Spearman correlation analysis to figure out the size of a file[11] and the number of method calls is a strong predictor. In general they state that building up a predictor is more powerful on the package than on the file level. Within their analysis

[11] in terms of LOC

the discovered average and maximum values for all metrics share a similar correlation to defects, whereas the total sum has the highest correlation. Above all they discovered that a defect model based on a previous release can be used to predict fault prone modules on a later release.
Kamei *et al.* [96] present work on Eclipse core platform revision 3.0 and 3.1. The authors claim a precision of 0,583 a recall of 0,179 and an F_1 of 0,274 with their hybrid model approach using LR and association rule mining.
Moser *et al.* [99] performed analysis upon the Eclipse bug dataset from the PROMISE repository [131]. Within their analysis change metrics (file and bug history, code chrun, *etc.*) clearly outperform static metrics (LOC, CC, *etc.*). The authors claim a 75% true positive rate and a recall >80% calculated using a 10 fold cross validation. They state a simple LOC only model might be good enough which represents similar findings like Ostrand *et al.* [78] and Bell *et al.* [82].
D'Ambros *et al.* [118] published an extensive performance comparison between different approach methods: process metric (code chrun, *etc.*), previous defects (bug history), static metrics (LOC, CC, *etc.*), Entropy of change (CC on code chrun) and Entropy. They used five public available datasets: Eclipse JDT Core, Eclipse PDE UI, Equinox framework, Mylyn and Apache Lucene summing up to 5885 classes and 1923 reported defects. They reproduced multiple defect classification experiments using their collected dataset and report AUC. Within their results classic code metrics (LOC, CC, *etc.*) perform best, followed by process metrics (code chrun, *etc.*).
Zhang *et al.* [129] presents work on mining 1.395 open source projects hosted on SourceForge and GoogleCode. Their aim has been to address CPFP building up a database of similarity. They introduced a four step approach including performing a partitioning, clustering, ranking and converting raw data. They identified six context factors (programming language, issue tracking, LOC, number of files, number of commits and number of developers) to define the clusters. Based on the correlation between projects metrics clusters are defined. Following up projects are assigned to those clusters. They performed SFP and CPFP upon their dataset reporting similar AUC values for SFP and CPFP demonstrating their universal defect predictor as a promising approach. Based on performance results the author state using all available data is beneficial when using machine learning algorithms, this is in line with Menzies *et al.* [88].

3.3.4. Summary

Within all the reviewed literature LOC is a strong predictor, CC and Halsteads metrics seem to be strongly correlated to each other and LOC. Process and change metrics tend to be better predictors for fault. Within industrial projects each individual software file seems to be developed by a single author thus author related metrics are less important than within open source projects with multiple contributors per file. The overall reported performance differs between the various publications. In general industrial SFP tend to report better performance than open source projects which seem to be related to more reliable fault report data.
The majority of published work prior to 2005 uses statistic methods (NBR, LR, Discrimant Statistic (DS), *etc.*). More recent work uses machine learning (NN, SVM, *etc.*) approaches. All reports using statistic models (*example given* Khosgoftaar *et al.* [62] or Ostrand *et al.* [78]) precisely report which metric to use and discuss upon their influence, machine learning uses maximum number of data (*example given* Menzies *et al.* [88]).
Table 3.2 hosts an overview to all reviewed case studies, more publications have been examined by Catal *et al.* [103] and Radjenović *et al.* [125] within their systematic literature surveys.

Table 3.2.: Overview to reviewed literature on SFP, I represents Industrial, OS Open Source

Publication	Type	Size	Projects	Predictor	Static metric	Change metric	process metric	social metric	performance	SFP/CPFP
[62]	I	1.3 Mill LOC	1	DS	*y*	*n*	*n*	*n*	68,9% TP	SFP
[67]	I	1.3 Mill LOC	1	GLM	*y*	*y*	*n*	*n*	*n* report	SFP
[73]	I	0.5 Mill LOC	3	NBR	*y*	*y*	*y*	*n*	*n* report	SFP
[78]	I	0.5 Mill LOC	2	NBR	*y*	*y*	*y*	*y*	83% TP on average	SFP
[82]	I	0.5 Mill LOC	2	NBR	*y*	*y*	*y*	*y*	63% TP on average	SFP
[85]	I	1.1 Mill LOC	5	PCA	*y*	*y*	*y*	*n*	low/medium correlation	SFP
[93]	OS	3.09 Mill LOC	3	LR	*y*	*y*	*n*	*n*	precision: up to 0.892	SFP
[96]	OS	9.726 classes	2	LR	*y*	*y*	*n*	*n*	precision: up to 0,583	SFP
[88]	I	119.000 LOC	11	OneR, J48, NB	*y*	*y*	*n*	*n*	71% TP	SFP
[94]	I	119.000 LOC	11	SVM, NN	*y*	*y*	*n*	*n*	87,4% TP	SFP
[100]	OS	119.000 LOC	11	SVM, ACO, LR	*y*	*y*	*n*	*n*	accuray up to 93,10%	SFP
[99]	OS	3.09 Mill LOC	3	OneR, J48, NB	*y*	*y*	*y*	*n*	75% TP	SFP
[111]	I	22.74 Mill LOC	12	LR	*y*	*y*	*n*	*n*	3,4% between projects	CPFP
[118]	OS	5885 classes	5	GLM, DT, NB	*y*	*y*	*y*	*n*	AUC 0,921	SFP
[129]	OS		1395	various ML	*y*	*y*	*y*	*y*	AUC 0,78	SFP

3.4. Public Available Datasets on Software Metrics

Within the last decade a number of datasets have been published containing software metrics, mainly gathered from open source projects. All these datasets contain at least static code metrics (LOC, CC, *etc.*) and bug information. Software revision history in general has been enabled by the usage of version control systems, e.g. Concurrent Versions System (CVS), SubVersioN (SVN), GIT, *etc.*. The authors of all available datasets crawled the revisions, measured the metrics and enriched the data with bug fix information from an Issue tracker or a ticket system. Most of the bug commit information has been extracted using the SZZ algorithm enhanced by Kim *et al.* [83] or manually annotated. A bug fix commit has either been introduced by the commit messages, *example given*"fixed bug", a link to a ticket or if two or three files have been changed as suggested by Kim *et al.* [83]. The majority of datasets come with a publication where the authors explain the datasets roots and may show an application with the data.
One of the first public available datasets has been released by the **NASA metric data program [133] (NASAMDP)**[12]. This datasource contains software metrics collected at ten different projects rooted within NASA flight software. This software has been written in C and heavily tested, all data can be considered as post release due to the spacecraft having already performed their missions.
PROMISE repository of software engineering data [13] has been founded and administrated by Sayyad *et al.* [79] and Menzies *et al.* [131]. It has originally started with the NASA Promise code repository and extend to 60 projects usable for SFP. It hosts most of the datasets published within the working conference on Mining Software Repositories (MSR)[14] conference series.
Software-artifact Infrastructure Repository (SIR) published by Do *et al.* [77] can be considered to be the first database on software bugs. It contains 81 projects with a rather small code size ranging from 24 LOC to 8.570 LOC. The programs are written in C, C++, C# and Java. All Bugs are hand seeded,

[12] http://mpd.ivv.nasa.gov

[13] http://promise.site.uottawa.ca/SERepository

[14] http://msrconf.org

which give them a high quality.
Kim *et al.* [83] introduce an algorithm to link a LOC containing a bug between its initial commit and its bug fixing commit. Their algorithms core uses *diff* (between the bug fixing and the prior commit) and *svn blame* after removing comments, blanks and format changes from the diff output. The authors claim to reduce the false positive rate up to 51% compared to previous algorithm. They validate their approach with data on *eclipse* and *Columba email client*. Noticeable in their report: ”Friday seems to be the most error prone commit day”, based on analysis from the eclipse commits.
Zimmerman *et al.* [93] present work on creating a bug database for Eclipse 2.0, 2.1 and 3.0. hosting 25.210 files with 25.585 defects. The authors claim to be one of the first to include data from the Issue tracking System and parse commit messages for ”bug fix”, ”bug”, *etc.*. The dataset contains traditional (LOC, CC, *etc.*) and various OO specific values (number of classes, number of method members, *etc.*). The authors differ between pre- and post release defects, only considering none trivial errors.
Kamei *et al.* [96] present work on Eclipse core platform revision 3.0 and 3.1. Their dataset consists of classic volume metrics (LOC), design metrics (CC) and OO metrics (methods per class, number of children, *etc.*). Their dataset contains 9.726 Java files of whom 16,98% are marked as faulty. The bug information is based on the developers ticket classification which has been retrieved via Bugzilla. The metric data has been measured using the Eclipse metric plugin.
Herraiz *et al.* [105] reports on an EU founded project collecting software metrics of 5.000 open source projects. Within their dataset description classic metrics (LOC, CC, process, chrun, *etc.*) are gathered along with issue tracking information. However there have been no recent reports on usage of this data within the software fault prediction community.
Mockus *et al.* [107] presents a dataset containing 1.398 projects hosted on GoogleCode and SourceForge. The authors state the set contains 207.904.557 files in total. As many open source projects recently switched their revision system the authors state that it is not a trivial task to maintain their dataset. This dataset has been showcased by Zhang *et al.* [129] building up their universal bug predictor.
D'Ambros *et al.* [113] uses a collected dataset on Bug reports from the Eclipse JDT Core, Eclipse PDE UI, Equinox framework, Mylyn and Apache Lucene projects. Their data has been collected between 1.1.2005 to 17.03.2009,

it contains software consisting of 2.131 classes and containing 1.923 bug commits.
Jus *et al.* [128] released a database system with the main focus on software testing research. The initial commit contains 357 bugs on five open source Java projects ranging between 22.000 and 96.000 LOC. The authors claim their database can be easily extended as it provides an API to add defect data and a full integration of SVN.

3.5. Software Fault Prediction

Software Fault Prediction (SFP) has been introduced in the late 1970is and early 1980is by work of Curtis *et al.* [55], Basili & Phillips [56] and Sherfi *et al.* [58]. They share a common idea to gather measurement data and guide engineers upon their decision where to look for bugs by rating the error sensitivity of single files or modules. According to historical analysis by Catal *et al.* [103] and Radjenović *et al.* [125] early SFP work focused on static software metric (LOC, CC, Halstead, *etc.*) and using statistical methods (LR, NBR, GLM, *etc.*) but reporting remarkable detection performance. Work past 2005 uses OO metrics (class level, coupling, fan in/out, *etc.*) and machine learning (SVM, C4.5/J48, RF, *etc.*.) as the predicting algorithm. According to **Radjenović *et al.* [125]** 68% out of 106 from their analysed studies used statistic models (see 3.5.1), 24% used machine learning (see 3.5.2) and 8% used correlation analysis. Upon Abaei and Selamats [127] impression too few works use OO related metric, even when analysing OO software.

3.5.1. Logistic Regression

Early work used statistic methods to derive probabilities for a file or module containing a bug. The majority of authors favoured either discriminant analysis, GLM or NBR. Common to all publications exists a vector[15] containing the measurement (software metric) data $\vec{x}_i$ of a file i and a desired result y_i representing either the number of fault or the probability of a file containing a bug. All authors split their dataset into a model fitting and an evaluation

[15] to cover consistency the original definition is adopted to fit this sections denotion

part. The majority of models are designed to describe 95% of the variance.

This section only analyses the prediction method. Additional information about the dataset, performance, *etc.* is presented within Section 3.3.1.

Khosgoftaar *et al.* [62] uses **nonparametric discriminant analysis**. Classic volume metrics (LOC, number of loops, number of if/else branches, *etc.*) along with design metrics (CC, number of calls, *etc.*) serve as input data after standardization to achieve a mean at zero and a variance of one. To overcome the deficit of correlated metrics[16] they used a stepwise discriminant PCA to extract uncorrelated variables. They define SFP as a two class problem, assigning fault free files to G_1 and faulty to G_2, see equation 3.4. π_i represents prior probability to being a member of G_i, $\vec{x}_i$ the vector containing the observations[17]

$$\hat{f}_k(\vec{x}_i|\lambda) = \frac{1}{n_k}\sum_{l=1}^{n_k}(\vec{x}_i|\vec{x}_{kl},\lambda) \tag{3.3}$$

$$y_i = \begin{cases} G_1 & \cdots if \frac{\hat{f}_i(\vec{x}_i)}{\hat{f}_i(\vec{d}_i)} > \frac{\pi_1}{\pi_2} \\ G_2 & \cdots otherwise \end{cases} \tag{3.4}$$

To tune their model they used a smothering factor λ. The model has been calculated splitting the series of $\vec{d}_i$ into a training and a test set. The authors loop until no improvement in the significance level can be observed. They report upon their evaluation the best model reports a type II misclassification of 3,4%, and a type I of 0%.

Graves *et al.* [67] reports on using **GLM** along with a logarithmic transformation of the metric values. In addition they introduce a files age as defined by the equation 3.5 where a_i represents the number of changed LOC and d_i the date of the change in years.

$$age = \frac{\sum_{i=1}^{n} a_i d_i}{\sum_{i=1}^{n} a_i} \tag{3.5}$$

[16] conform Section 3.2.1

[17] the measured metric data

They derive various models consisting of different metric types. Their best performance is of the model described within equation 3.6, denoted as the model G.

$$\frac{7}{100} log(\frac{LOC}{10^3}) + \frac{95}{100} log(\frac{a_i}{10^3}) - \frac{44}{100} age \tag{3.6}$$

Within their analysis a files age is a very strong indicator for faultprones. The factor $\frac{44}{100}$ states a exp $\frac{44}{100} = 0,64$ reduced rate of having an error if the change occurred in a file containing the same amount of deltas (d_i) a year earlier.
Ostrand *et al.* [78] uses **NBR** to predict the number of faults within one file during a release. They assume a poisson distribution among y_i with mean λ_i as defined by equation 3.7 where $\vec{\beta}$ represents the regression coefficients to x_i and λ_i as a random number drawn from a gamma distribution having a mean of 1 and a σ^2 ‘ 0 representing the dispersion coefficient to model the fault dispersion. β and σ have been tuned using maximum likelihood.

$$\lambda_i = \gamma_i e^{\vec{\beta}^T x_i} \tag{3.7}$$

3.5.2. Machine Learning

The first application of machine learning onto SFP has been reported by Khoshgoftaar *et al.* [63] using Neural Network (NN) to predict failures within a Telecom software written in PROTEL[18] consisting of 7 Million LOC, organized within 7.000 modules, 14% of them contain more than three faults. Their metric input consists of design metrics (CC, if/else statements, loops, procedure calls, *etc.*). All metric values have been normalized to mean zero & variance one. Domain metrics have been derived using a PCA. The full dataset has been split into fit & test (used for DS) and train & validate (used for NN) sets. The DS has been designed using equation 3.4 [19]. Their NN uses a feed forward three layer perceptron network with four inputs, two[20] outputs and a supervised backpropagation training. According to the authors' analysis the NN performs slightly better than the DS: recall 0,95 to 0,92, precision 0,86 to 0,88 and accuracy identical at 0,86 but has an overall

[18] high level language comparable to Pascal

[19] described at Khosgoftaar *et al.* [62], see Section 3.5.1

[20] one for fault and one for none fault prone class

misclassification error of 0,262 to 0,295. They report upon challenges due to the low number of bugs at 14% which requested multiple remodels upon the NN.

Guo *et al.* [76] uses the NASAMDP dataset to evaluate the applicability of Random Forest (RF) to SFP. Within their analysis RF are well suited due to the scalability to large datasets and its ability to report the importance of single attributes like the metric values. To overcome the drawback of RF to minimize the overall error, which would result in a high misclassification in case of a small number of faulty files, the authors adopt the C_{utoff} value. They report an accuracy of up to 0,94 and a true positive of up to 0,87. Due to the small dataset they used a 10 fold cross validation to validate the results. Among the attribute analysis LOC and Halstead (effort and volume) are the strongest metrics related to faults.

Menzies *et al.* [74] and [88] has been among the first to use machine learning on fault prediction. They evaluated OneR, J48 and NB using the NASAMDP dataset. They achieved the best performance using NB with logarithmic metrics.

Mizuno *et al.* [115] presents work on Eclipse BIRT and TPTP plugins using NB, J48, LR, NNge[21]. Their analysed software contains 10.389 modules with roughly 40% faulty ones. Their experiments present NB as the best predicting algorithm archiving a precision up to 0,570 and a recall 0,947 when using the previously presented SPAM filter based metric P_{TPF}.

Recent work on this topic by He *et al.* [130] where they compare multiple predictors (J48, LR, NB, Decision Table, SVM and Bayesian Network) using ten projects from the PROMISE [131] achieve with overall 33%[22] of defective files. In their analysis NB has the highest predictive power with a precision of 0,49, a recall of 0,588 and an F-measure of 0,496[23]. The authors evaluated the impact upon metric usage between the full set and a selection of five metrics. According to their analysis there is a slight decrease in the predictive power in terms of precision, recall and F-measure, thus the authors state that the slight performance reduction might be acceptable as the model is much simpler.

Common to the majority of recent machine learning approaches the WEKA

[21] Nearest-neighbor-like algorithm using non-nested generalized exemplars

[22] minimum 3.5% and maximum 75%

[23] all performance is reported as median on all ten projects

toolkit [104] is used, which reduces the impact of implementation errors and enabled performance comparison between various publications. A highly favoured dataset is the NASAMDP which was used within multiple reports. Among the result reports NB and Random trees perform best (Guo *et al.* [76], Mizuno *et al.* [115]), even within rather simple metric settings (He *et al.* [130]). The majority of publications favour the usage of as many as possible attributes when using machine learning approaches (Menzies *et al.* [88]), but the decreased predictive power when using a simple metric set might be very narrow (He *et al.* [130]).

3.5.3. Performance

Various studies have been released reporting on SFP performance. The majority of reports use AUC, recall, precision, True Positive (TP) and False Positive (FP) as defined within the equation 3.9. There is only the NASAMDP dataset KC1 used across multiple publications, where the majority reports on the AUC[1] value, see Table 3.3. Other values are given too, but can not be compared easily as they do not occur within all studies.
A visual explanation defines TP as the number of correctly identified bugs, True Negative (TN) as the number of correctly identified fault free files. In contrast FP and False Negative (FN) represent the opposite where fault free files are classified as bugs and vice versa. The recall figure represent the share of correctly found bugs. Precision shows the share of correctly classified bugs out of all reported bugs, F1 is the harmonic mean between precision and recall. Type I and II represents the share of FP respective FN out of all reported classes. AUC[1] gives an estimate of the predictor's performance compared to a random chooser. The ideal predictor shall have AUC equal to one, the random predictor is supposed to report 0,5.

$$AUC = \frac{1}{m \quad n} \sum_{i=1}^{m} \sum_{j=1}^{n} 1_{p_i > p_j} \tag{3.8}$$

[1] see equation 3.8

Where:

$$\begin{aligned} m &: \text{the true datapoints} \\ n &: \text{the false datapoints} \\ p_i &: \text{probability by classifier for true class} \\ p_j &: \text{probability by classifier for false class} \end{aligned}$$

Ostrand and Weyuker [90] discuss performance results from their previous work regarding Bell *et al.* [82] and Ostrand *et al.* [73]. The authors discuss performance report values and effects causing them to lead to false interpretation. If a dataset contains a very low number of bugs, accuracy[2] and recall[3] might be close to 100%, but still the predictor might miss the faulty files. In such a case the type II misclassification rate[4] is near zero.
G-Measure, see Equation 3.9h, is defined by Jiang *et al.* [134] to adjust sensitivity in case of imbalanced class problems and still present a measure to compare different classifiers similar to F1. In an ideal setting the G-Measure is one. Weiss [126] favours F1 and G-Measure when dealing with imbalanced class distributions, as all other measures (AUC, accuracy, $\cdots$) report delusive figures due to a high number of TN rooted in the class distribution.

[2] see equation 3.9a

[3] see equation 3.9b

[4] see equation 3.9e

[5] see equation 3.9f

[6] see equation 3.9c

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.9a}$$

$$recall = \frac{TP}{TP + FN} \tag{3.9b}$$

$$precision = \frac{TP}{TP + FP} \tag{3.9c}$$

$$TypeI = \frac{FP}{TP + TN + FP + FN} \tag{3.9d}$$

$$TypeII = \frac{FN}{TP + TN + FP + FN} \tag{3.9e}$$

$$F1 = 2\frac{precision \cdot recall}{precision + reall} \tag{3.9f}$$

$$pf = \frac{FP}{TN + FP} \tag{3.9g}$$

$$G - Measure = 2 \cdot \frac{recall \cdot (1 - pf)}{recall + (1 - pf)} \tag{3.9h}$$

In their opinion a report on a good working fault predictor should contain a high accuracy[2], a high recall[3] and a low type II misclassification[4] rate. The authors did not present F_1[5] which represents the harmonic mean between precision[6] and recall[3]. Ideally F_1 is one for a perfect prediction.

Moser *et al.* [99] presents work on Zimmerman *et al.* [93] eclipse dataset stating that change metrics (commit size, file age, refactoring, *etc.*) clearly outperform static metrics (LOC, ...). They are using J48 as the predicting algorithm and achieve an accuracy[2] of up to 0,87 and a recall of up to 0,8.
Lessmann *et al.* [97] compared the predictive performance of six categories (statistic, nearest neighbour, NN, SVM, decision tree and ensemble) of classification algorithm in eleven projects from the NASAMDP dataset supplying the available metrics (volume, design and change). The majority of predictors perform similarly and there is no single method which has a performance superior over all datasets. Analysing their data one can see a stronger scattering among the projects than across the learners, thus choosing the prediction algorithm is less important.
Catal *et al.* [102] evaluates prediction performance in different (sub)selections

of metrics and machine learning predictors evaluated on the NASAMDP dataset. Within their analysis RF and NB performs best with an AUC between 0,79 and 0,84. Table 3.3 presents the reported AUC values from their experiments in the KC1 project. Using the full metric set (LOC, CC, CK) performs best (see Experiment 1), only using a subset (see experiment 2 or 4) performs slightly lower. The best overall performance (0.89 AUC) on the KC1 dataset has been achieved by Singh *et al.* [109] using SVM and all available metrics.

D'Ambros *et al.* [113] presents an intensive study upon predicting performance based six different metric categories (change, bug history, static volume, code chrun, entropy of change & static volume and combined approaches). The authors report about the Spearman correlation between predicted and real bugs. Clearly the combined approaches outperform the single metrics styles. This leads to a consistent statement with Jiang *et al.* [95] and Menzis *et al.* [88] to achieve best performance when combining the maximum number of metrics.

Menzis *et al.* [135] and Menzis *et al.* [114] introduce the "ceiling effect", stating no performance improvements over the last couple of years as observed by nine publications they studied, including Lessmann *et al.* [97]. All their meta studies report upon AUC values up to peak 0,9 on the same NASAMDP dataset. Menzis *et al.* suggests shifting the performance reports from classic values (AUC, TN, TP, precision, recall, F-measure) to finding most faults in the smallest number of modules. Rahman *et al.* [120] uses the AUCEC performance figure, stating that this is the most cost sensitive AUC. The calculation is similar to traditional AUC, but the defective files are ranked according to the predicted defect sensitivity. The idea is to define a measure ajar finding the most bugs with the lowest number on files to inspect. This performance value represents the ability to predict bugs when a tester is only able to inspect a defined share of files or LOC. This scenario might be represented using AUCEC20, stating the predictive power if a tester is only able to inspect 20% of all LOC.

Abaei and Selamat [127] report of prediction performance using accuracy[24], recall[25] and precision[26]. It is one of the view works where reviewing au-

[24] see equation 3.9a

[25] see equation 3.9b

[26] see equation 3.9c

Table 3.3.: AUC performance results on NASAMDP project KC1, empty cells represent not reported values by the publications

KC1 (AUC)							
Algorithms	Catal *et al.* [102] # 1 - 21 Metrics	Catal *et al.* [102] # 2 - 13 Metrics	Catal *et al.* [102] # 3 - 21 Metrics	Catal *et al.* [102] # 4 - KC Metrics	Abaei *et al.* [127]	Mende *et al.* [106]	Singh *et al.* [109]
AIRS1	0,6	0,59	0,6	0,7	0,563		
AIRS2	0,57	0,59	0,6	0,71	0,529		
AIRS2Parallel	0,61	0,6	0,6	0,68	0,605		
Bootstrap aggregating						**0,82**	
CART decision tree						0,69	
CLONALG	0,52	0,53	0,52	0,67	0,532		
Decision Table					0,785		
Decision tree, J48					0,689		
Immunos1	0,68	0,71	0,68	0,69	0,681		
Immunos2	0,51	0,5	0,49	0,72	0,511		
J48	0,7	0,7	0,7	0,75			
GLM						0,81	
NB	**0,79**	**0,79**	**0,8**	0,76	**0,79**	**0,79**	
RandomForests	**0,79**	**0,8**	**0,79**	**0,79**	**0,789**	**0,84**	
SVM							**0,89**

thors have performed experiments on their own and one can rely on the comparability of performance report values.
Overall all performance studies report of a high prediction ratio when using a combination of multiple metric suits. AUC values up to 0,89 (Catal *et al.* [102]) or accuracy up to 0,87 (Moser *et al.* [99]) have been reported to be reached with machine learning approaches. The numeric performance results between various machine learning algorithm differ slightly, RF and NB tend to be the best performing (Catal *et al.* [102]), SVM are achieving similar or slightly better results (Singh *et al.* [109]). There is only one dataset used across multiple studies (NASAMDP KC1) and several studies comparing performance (D'Ambros *et al.* [113]) with its own implementations.

3.5.4. Summary

Software Fault Prediction (SFP) utilizes metric data to predict a component's (method, module, source file, *etc.*) failure using a predicting algorithm. There are multiple literature reviews (Catal *et al.* [103] and Radjenović *et al.* [125]) detecting an increased number of publications after 2005 when the NASAMDP was released. Early work (Khosgoftaar *et al.* [62], Ostrand *et al.* [78]) uses statistic methods DS, GLM LR, NBR, *etc.*, later (Menzies *et al.* [74], D'Ambros *et al.* [113],He *et al.* [130]) use machine learning NB, SVM, RF, *etc.* mostly using the WEKA [104] toolkit. Various software metrics were introduced, starting with simple volume as Lines Of Code (LOC) or Halstead [54] and design such as Cyclomatic Complexity by McCabe (CC) [53] continued by change (code chrun, file and failure history) and measures like Chidamber Kemerer [60] OO metric (CK). The majority of work addresses a strong predictive strength to code chrun and file & bug history (Ostrand *et al.* [78], He *et al.* [130]) with empirical evidence by Tomaszewski and Damm [86] where changed classes or LOC show a 20 to 40 times higher chance of introducing a failure than newly added classes. Early SFP work discuss fine grained which metric to use, machine learning approaches use the maximum available number of metrics, whereas some reports (He *et al.* [130]) indicate a minor loss in predictive output accuracy if reducing the number of metrics in use. Performance reports use recall (equation 3.9b), precision (equation 3.9c) and accuracy (equation 3.9a) to report upon the quality of predictive results. The majority of machine learning approaches (D'Ambros *et al.* [113], He *et al.* [130]) is able to archive precision and recall beyond 0,8. Lessmann *et al.* [97] argues the choice of predicting algorithm is less important which is in contrast to Menzis *et al.* [114] who states tuning algorithm has a significant influence on the achievable predictive performance.

3.6. Cross Project Fault Prediction

The previously introduced Software Fault Prediction (SFP) relies on the existence of historic metric data. In the early phases of a project this historic data might be too short. Cross-Project Fault Prediction (CPFP) tries to overcome this deficit by using available metric data to predict failures within another (new) project. During this section SFP is used as a synonym for predicting faults within the same project and CPFP to train from multiple projects and predict within a target project.

3.6.1. Selected Publication

First work on this topic has been presented by **Briand *et al.* [71]** where they study two commercial developed Java OO: software products xPose and Jwriter which have been developed at Oracle Brazil. The dataset contains 212 classes and 2,767 methods in total. The authors describe using xPose as the training set, as it is the bigger system and Jwriter as the evaluation dataset. As using OO software they collected CK and classic volume LOC metric. They used two predicting algorithms, LR and Multivariate Adaptive Regression Splines (MARS). Their overall performance using LR is reported on recall at 0,45, precision at 0,737 and accuracy at 0,85, compared to the MARS model which performs slightly better in terms of recall at 0,48 but lower on precision at 0,68 but comparable at accuracy by 0,84. Still this means the model misses 40% of the faulty files. The author states that transferring a fault prediction model between two projects can not be considered as straight forward, even when the development environment in terms of coding guidelines, tool usage and company settings is very similar.

Zimmermann *et al.* [111] performed a fault prediction transfer study on a large scale using four Open Source projects (Apache Tomcat, Apache Derby, Eclipse and Firefox) and eight closed source projects developed by Microsoft (Direct-X, IIs, Printing, Windows Clustering, Windows File system, SQL Server 2005 and Windows Kernel) summing up to 35 Million LOC. They collected volume metric (LOC, CC, *etc.*), change metric (code chrun, *etc.*) and process metric (number of developers, commit and file and bug history, *etc.*) which have been used as normalized values together with a LR based predictor. Between their twelve projects they performed 622 cross project experiments. A successful prediction is counted if recall, precision and accuracy are above 0,75. This lead to a very weak success rate at 3,4 %. Upon their analysis results some projects tend to be more related than others but the prediction might not be bilateral. The authors define a similarity vector between two projects to derived a decision tree if cross project prediction is possible. Their core finding is related to Briand *et al.* [71] stating that similar setting between two projects are beneficial to CPFP.
Turhan *et al.* [110] presents work on seven projects from NASAMDP and three industrial, all hosted at PROMISE archive. They used classic metrics (volume and design) as supplied by the projects and NB as the predictor. They have chosen to use 90% randomly selected data from the project metrics to train the predictor and the remaining 10% to test their learned model. Their work aims to compare SFP with CPFP and perform an in deep analysis upon the predictive performance. Overall they report on high prediction in CPFP, stating the probability to detect an error goes up to 97% at the cost of probability for false alarms of up to 100%. In their opinion this are the best ever reported values, but make the system practically unusable due to the high number of false alarms. Upon their analysis this effect is caused by the high number of training samples they can provide when applying CPFP, which leads to their discovery that a too high number of training samples increases the variance within the prediction model causing the high false alarm rate. Second they stress upon their previous findings to use all available data. Using Infogain ranking among all collected metric data they discovered similar correlation between single metric values and the number of bugs, stating there is no perfect metric. To overcome these effects upon the too large training set, they applied nearest neighbour filtering to select a maximum of 200 training samples, which decreased the false positive to 60-65%. Thus they argue that with CPFP it is important

to select suitable training data between the target project and the training projects. Overall they conclude SFP still to be the best prediction, but using the nearest neighbour to select a maximum of 200 training samples is better than using the full raw data to perform CPFP.)

Rahman *et al.* [120] claims that CPFP has a similar performance to SFP when measuring the AUCEC value. They present work on nine Apache projects yielding CPFP can identify the most fault prone modules but might still give a high false positive rate. In their definition it is more important to find some bugs within a strict limited inspection budget, which is represented by AUCEC. Their case study uses classic change metrics, number of commits, number of developers, code chrun, number of new features, number of improvements and classic volume metric LOC. Their model has been set up using LR and all available metrics data as suggested by Menzies *et al.* [88]. Following their definition a file is set as defective if there is at least one defect report within the corresponding release.

Herbold [122] presents work following Turhan *et al.* [110] on enhancing CPFP by filtering the training data. Herbold selected clustering using the EM algorithm and nearest neighbour along with a weight schema to select training. He evaluated his system using thirteen open source projects written in Java with an average of 34% defective classes per project. The dataset is available to the public via the PROMISE [131] repository. To predict failures he used twenty standard static metrics (LOC, CC, code chrun, *etc.*) and seven machine learning (LR, NB, Bayes Network, SVM with an radial basis function kernel, C45, RF and a multilayer Perceptron) predictors. He defines a successful prediction if recall is $\geq$ 0,7 and precision $\geq$ 0,5 which is used to report on the success rate, see equation 3.10

$$sucess\ rate = \frac{number\ of\ sucessfull\ experiments}{number\ of\ experiments} \tag{3.10}$$

The training data is filtered using the EM clustering and the nearest neighbour algorithm which leads to an increase in success rate from 0,09 to 0,18 when using CPFP. This is good, but still low compared to SFP with a success rate of only 0,37. Within their analysis the discovered SVM is said to be the worst predictor but after applying the training data selection however, SVM performs the best. Overall Herbold suggests using the nearest neighbour algorithm with a 50% - 70% neighbourhood when performing CPFP.

Zhang *et al.* **[129]** present a unique approach to CPFP by building up a database with similarity index between projects. If two projects are similar, the learned predictor can be reused, which leads to a universal defect predictor if the database contains enough datasets. During their evaluation they gathered data from 1.398 projects hosted at SourceForge.net and Google code containing between 20% and 60% defective files. More than 60% of the datasets projects apply to issue tracking, which forces the authors to classify defective data based on a regular expression statement on the commit messages. They collected classic static metrics (LOC, CC, file and fault history, code chrun, *etc.*) and CK metrics. The authors claim to achieve 70% of successful predictions using the same criteria as Herbold [122] ($\geq$ 0,7 and precision $\geq$ 0,5 and equation 3.10). Their context aware rank transformation approach tries to overcome the observed distribution of variance between different project contexts. Every projects metrics set is: 1.) partitioned into six none overlapping context groups[27], 2.) clustered 3.) ranking function is derived 4.) the ranking function is applied, which ensures values to be at the same scale. Every project has been tested using SFP with the ranking function against a classic log model evaluated using a ten fold cross validation. The authors report on similar to better performances in terms of AUC, recall and precision, using their ranked transformation with the cost of a higher false positive rate.

He *et al.* **[130]** gathered metric data from ten open source projects from the PROMISE archive. Within their setting they used metrics collected from previous releases of the same project (refer to their scenario two) or from different projects (refer to their scenario three). Upon their analysis the best predictive algorithm is different, Decision Tables perform best at recall 0,674 and precision 0,549. Compared to their SFP analysis the reduced metric set performs 10-15% lower than the full set. Overall the predictive performance between the ”within project fault prediction”, even across multiple releases, has got a more similar sound than the CPFP across all ten analysed projects. The overall predictive power is lower on CPFP than on SFP.

[27] programming language, issue tracking in use, LOC, number of files, number of commits, number of developers

3.6.2. Summary

The overall reported performance on CPFP is significantly lower than on SFP when evaluated on the same dataset. Performance values range between Briand *et al.* [71], where they miss more than 40% of faults up to Zhang *et al.* [129] who claims to predict up to 70% of all faults. Large scale experiments by Zimmermann *et al.* [111] and Zhang *et al.* [129] state that CPFP benefits from similar project settings (tools, development guidelines, *etc.*). Analysis by Turhan *et al.* [110] states using too big training data harms the performance. Work by Turhan *et al.* [110] indicates pre filtering training data before applying CPFP to increase predictive performance. This has been evaluated by Herbold [122] who reports upon doubling the success rate when using nearest neighbour filtering on the training data. He *et al.* [119] perform a similar study supporting Turhan *et al.* [110] and Herbold [122] observation regarding the selection of training data for successful CPFP. The majority of studies uses data from PROMISE and NASAMDP archives which offer standard static metrics (LOC, CC, code chrun, *etc.*). Similar to Section 6.1 NB, RF and SVM are the dominating predictors, but the predictive performance differs only slightly.

3.7. Imbalanced Class Distribution

If the classes within a dataset are not equally distributed this is known as ”imbalanced class problem”. Weiss [126] differs between low (1:10), mid (1:100) and strong (1:1000 and beyond) imbalanced datasets. The author argues multiple classifiers having problems with learning from the minority class or even over fitting them. He outlines ”divide and conquer” based algorithms, such as SVM or trees, might lead to fragmented data. It might happen as single leaves or class split decisions are based on a minority sample which might not be representative, thus it is advised not to use them on such a dataset.

Japkowicz [69] presents under- and oversampling as methods to overcome the imbalanced class problem. The author study an artificial dataset with binary class distribution. Upon their results both strategies are valid to increase the predictive performance.

Drummond *et al.* [75] is among the first to address influences of minority classes when applying C4.5 learners. They suggest to use undersampling to overcome this deficits.
Khoshgoftaar [116] presents work on an Eclipse dataset dealing with imbalanced class distribution. They focus on reducing the number of input dimensions (software metrics) by applying feature ranking. Within their analysis the reduced set performs better than the full set. In addition they discovered sampling the data has no significant impact on the resulting performance.
An empirical evaluation by Posnett *et al.* [117] reports on problems when dealing with imbalanced class distributions. Their applied machine learning algorithms can not achieve good performance.

3.8. Software Error Analysis

Not every bug leads to a failure and not all failures are severe. Various reports studied the roots of bugs and techniques to prevent the genesis of them. This Section presents some of these reports and guidelines.

3.8.1. Selected Publication

Adams [57] conducted a survey on large IBM software products between 1975 and 1980 collecting issue reports and the associated bug fixes. He discovered that large portions of failures never lead to software errors, but less than 2% of the known faults cause the significant errors in operation. This work is mainly based on discovering the probability for defect to occur. If a bug needs hundreds of thousands of hours to occur, the testing team might not be able to discover them all. Most of the none discovered defects occur only on rare system settings. The author claims that service is necessary for every software product.
Fenton and Neil [65] and Fenton and Ohlsson [66] present eight theses, four of them with empirical evidence, concerning the occurrence of software faults. They state that the majority of faults are discovered within a small number of modules this is supported by Denaro and Pezze's [72] and Bell *et al.*'s[82] work. Secondly they state that the fault density broadly stays similar during the development and testing stages across different projects

if they are developed within the same environment (company, tools, *etc.*). Further fault densities are comparable among multiple releases for similar testing and operation phases.

Boehm and Basili [70] released their top ten fault reduction list compiled upon their experience and prior empirical work. They state that fixing bugs later costs more, 80% of revision is caused by 20% of the bugs although 50% of files might even be bug free and up to 90% of downtime are caused by only 10% of the bugs. In their opinion disciplined developers can reduce the risk of defect introduction by 75%. Not all fault prevention methods work on different bugs, multiple of them guard the same error class (*example given* clean room development and peer review) leading to a potential save on money without quality drawbacks.

Wagner *et al.* [81] present work on types of bugs discovered by static code checkers, testing and manual code inspection. They performed a case study on five industrial projects from the Telecom sector and one academic. All software is written in Java realizing web information systems, in total those five projects consisting of 1882 classes and 123.000 LOC. In conclusion, they state that static analysers find a subset of bugs discovered by code reviews, dynamic tests finds completely different bugs. This means there is no overlap on bugs discovered using these two methods which makes them complement each other. The authors note that static analysis tools only search for specific patterns which might trigger false alarms, *example given* if a developer uses two different methods upon opening or closing a database connection.

Vipindeep and Jalote [80] present a literature survey concerning well known programming failures and techniques to prevent them. They categorize bugs as *Catastrophic*, a system may crash or lose functionality or expose security issues, *Major*, malfunction or loss of data, *Minor*, displaying information in the wrong format and no effect like typographic errors, however they do not present statistics upon their occurrence.

Li *et al.* [84] analyse bug fix traditions of open source software by reviewing and categorizing 29.000 Bug reports from Open source software. The reviewed software Mozilla and Apache consist of 4 Million LOC in total. Overall the authors conclude that bugs are different than within the past ten years. One insight concerns less memory related bugs which may be due to the recent heavy usage of leak checkers. Null pointer errors are still common which could be prevented by static code checkers. Semantic

bugs which range from missing cases or features, to wrong control flows and false data processing to simple typos in messages. They account for 81,1% - 86,7% of all bugs which represent a major problem. The number of discovered security related issues rises as there are more attacks on software. The software's core functionality and Graphical User Interface (GUI) area are the most error prone regions, but the most fault causes lead to incorrect functionality (75,2%), minor to crashes (13%) or hang ups (2,5%).
Boogerd and Moonen [45] present work on MISRA 2004 rule violations and the coding standards ability to prevent faults. They conducted an industrial case story about a Secure Digital Driver software developed at NXP where they queried the issue tracking and version repository to establish a link between fault and rule violation. The studied *software* did not use MISRA checks during development, thus it is possible to establish a link between faults and rule violation[28]. The authors are using LR to analyse correlation between faults and rule violations. Upon their findings there is a strong positive correlation between bugs and rules violations for 22 out of 72 MISRA rules. Compared to randomly selected files the rule checks are better fault predictors. Rule violations are discovered using static code analysers, commonly suffering from a too high false positive alert rate. The authors state that a too high number of reported rule violations can cause software to be less reliable as developers tend to ignore them. Thus selecting the correct subset of rules is important.
Boogerd and Moonen [101] continue their work with another embedded software by NXP representing a mature television platform system. Comparable to the other study they identify ten out of 88 MISRA rules to show a positive correlation with faults reported within the issue tracking system. In contrast the correlating rules are different expect for one. As the two analysed software projects are very different the authors conclude that rule violations and bug correlation is domain dependant. Further files with a higher rule violation density tend to be more error prone, at least regarding ten MISRA rules. The authors do not present characteristic of the use case stories code (LOC, CC, structure, etc.). As static code checkers produce a high rate of noise due to false alarms the authors suggest starting with a minimal set of rules and add new rules during development. Illustrating

[28] if developers use MISRA checks they have to fix violations prior to testing stage which influence the correlation analysis

this, they state that 30% of their used case software LOC trigger a rule violation warning.

3.8.2. Summary

Their exist multiple works stating that only 20% of the files/modules contain 80% of the bug, compare Bell *et al.* [82] and Menzies *et al.* [88]. Adams [57] state that less than 2% of the known bugs cause significant problems. Similar Li *et al.* [84] discovered that a small amount of faults (15,5%) lead to crashes or software hang ups. Boogerd and Moonen [45], [101] discovered a positive correlation between the number of bugs and the coding standard rule violation density within files, suggesting a rule violation might be a useful fault predictor, but the selection of fault indicating rules scatter with the projects domain. Wagner *et al.* [81] back the usage of static code analysers and dynamic tests as complement methods. There are no reports concerning the type of faults occurring within industrial projects or empirical evidence of the fault preventing power of coding standards. Wagner *et al.* [81] give limited insights into this by the error classes' static code checkers identify. Boehm [59] and Vipindeep and Jalote [80] present common occurring faults without listing empirical statistic.

4. Development Tools and Methods used within the Automotive Industry

This chapter presents an overview of the tools and methods used during the automotive software development process. The presentation starts at the specification stage and finishes when the software is released to customers. The presented content is based on work by Altinger *et al.* [13], Altinger *et al.* [22][1]and Altinger *et al.* [23].

4.1. Questionnaire Survey

Upon best knowledge no reliable data in literature about tools used within the automotive domain, refer to Section 3.1, therefore a questionnaire survey has been conducted. This survey consists of 24 questions and covers 68 responses from developers within the automotive industry class divided into Research (RU), Pre Development (PD) and Series Development (SD). The questions can be extracted from Appendix B or online [136].

4.1.1. Survey Setup and Meta Data

Initially the questions were distributed to 45 persons related to testing and requirements engineering within the automotive software industry located in Research and Development (RD) departments. All of them are personally known to the authors, resulting within the following distribution: OEM 53%

[1] This publication is submitted to review and is not published at date of release of this thesis

[2], Supplier (Tier 1-3), engineering service provider, software vendor (SES) 40% [3], university/research facility 7% [4]. All receivers were asked to forward this survey to others, thus it is not possible to calculate the response rate. Achieving a representative poll was non an objective of this survey, the main aim was to gain insight into used tools and methods when testing software. The survey was conducted anonymously, thus it is not possible to eliminate duplicated answers from identical departments. The questions were accessible online between 1^{st} of February and 4^{th} of April 2014. In total 68 responses were given resulting in the following distribution: OEM 37,74%, SES 52,83%, University / Research facility 9,43%. 41,1% of whom are software developers, 9,6% requirement engineers, 20,5% test engineers, 23,3% managers and 5,5%others (mainly researchers). 17,92% of respondents belong to RU, 42,45% to PD and 39,62% to SD. All questions offer multiple choice with the ability to tick all options or state a free text answer.

4.1.2. Survey Results

The results of the analysis are conducted among three groups: *testing*, *specifications* and *personal*. All interpretation and Figures 4.1 to 4.12 are split into three departmental categories: Series Development (SD), Pre Development (PD) and Research (RU). Following the explanation by Crolla [29] and Altinger *et al.* [13] RU performs research in which the main aim is to investigate algorithms and innovations, they may present their work by proof of concepts, displaying limited functions, not considering any limitations by embedded systems, regulatory laws, *etc.* PD uses the output of RU and adopts their findings to automotive limitations *example given* sensors field of view, limited CPU power, environmental hazards like weather conditions *etc.* They may also conclude their work by proof of concept demonstrations, however, they may also consider norms *example given* ISO26262 [49]. SD is responsible in developing the actual car, implying they have to carry out the majority of tests and consider all safety and reliability requirements defined by either the company or regulatory laws and norms.

[2] 10 companies: AUDI, BMW, Daimler, Fiat, Ford, Mitsubishi, Scania, Toyota, Volvo, VW

[3] 16 companies, among them Bosch, Continental and Magna

[4] 3 universities: KTH Royal Institute of Technology, Graz University of Technology, Frauenhofer ESK

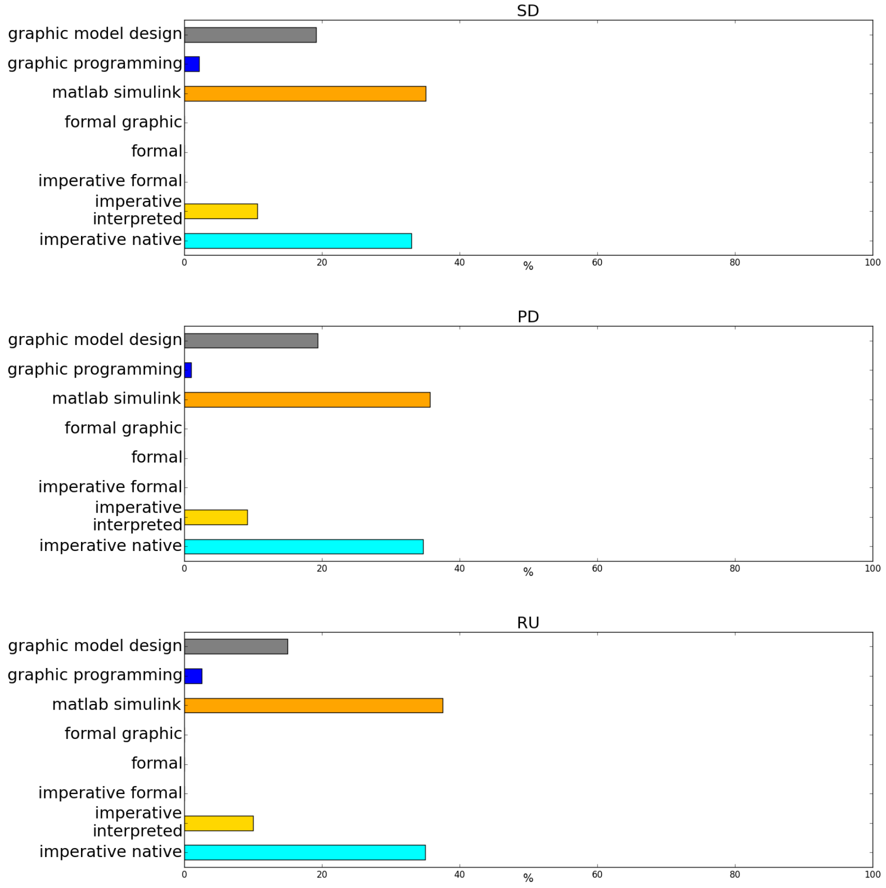


Figure 4.1.: Languages used as a result of a Survey by Altinger *et al.* [13]

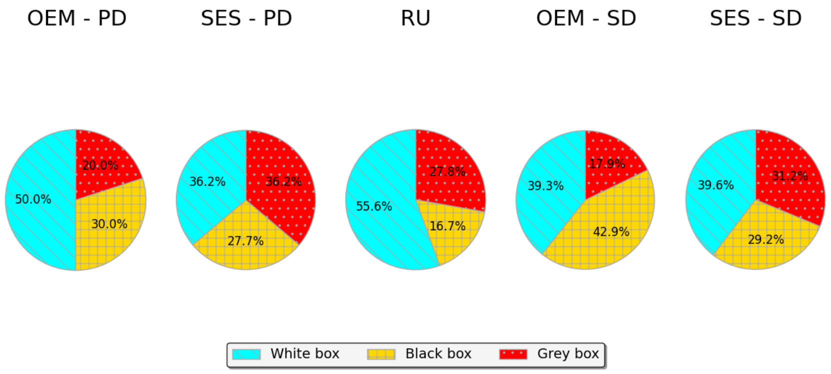


Figure 4.2.: Code Access by RD departments as a result of a Survey by Altinger *et al.* [13]

Collective to all three department types Matlab Simulink [41] is the most common tool to design models, refer Figure 4.1. These findings match work by Broy *et al.* [31] and Bock *et al.* [38]. A similar share goes to imperative native languages like C and C++.

Adjacent to the programming language, access to the source code or model is requested. As automotive development is distributed between multiple companies, see Broy [7], Figure 4.2 adds the terms OEM and SES. Blackbox denotes no knowledge of the implementation including source code, algorithms, state-machines, *etc.* The tester may only know the interfaces and requirements as defined when instructing the software development. Whitebox denotes full access to the source code, Grey box represents knowledge about algorithm and coding structures but not to the actual source code. PD and RU received the highest access rate to source code, in contrast SD got the highest Blackbox testing, noteworthy SES got more access to structures and algorithms where the majority of OEM SD departments perform Blackbox tests based on the specification, this matches statements by Crolla [29] where SES develops the software and OEM only performs the final testing based on the specifications written within the requirements document.

As explicitly stated within Section 2.2, 4.2 and Figure 2.2 the specifications are a core part of the development process. Figure 4.3 presents the usage of

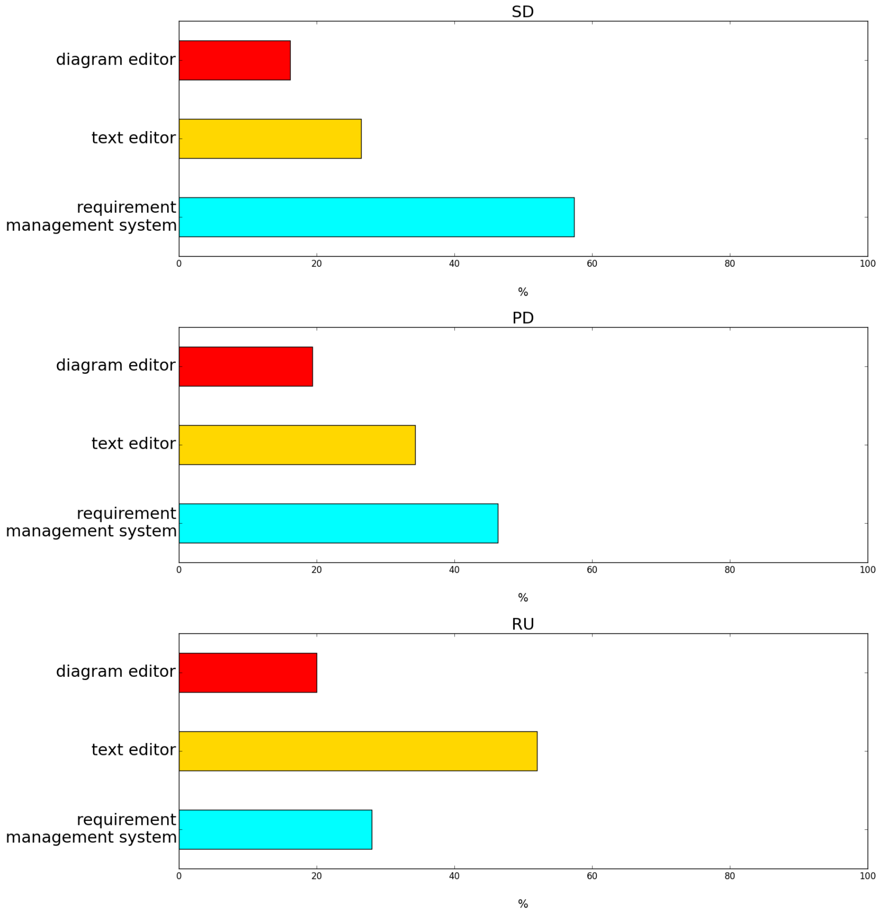


Figure 4.3.: Tool categories to write specifications as a result of a Survey by Altinger *et al.* [13]

Table 4.1.: tools in use to write specifications

name of tool	vendor	[%]	citation
Rational Doors	IBM	38	[137]
Word	Microsoft	17	
Integrity	PTC	13	[138]
Excel	Microsoft	11	
Enterprise Architect	SparxSystems	7	[139]

”requirement management systems” as the dominating tool category within PD and SD which matches the prominent tools listed in Table 4.1. According to the survey IBM Doors [137] is the most common tool with which to write specifications, however the Microsoft Office Suite (MsOffice) family is the second most influential, which belongs to the ”text editor” tool category. Noteworthy RU prefers text and diagram editors which may match the department’s definition which does not require tracking and fulfilling all norms and laws.
IBM Doors [137] is a database tool to write and track specifications. It is possible to state textual and graphical requirements, organize them hierarchically or link them to matching specifications. Editing authors are traced, resulting documents can be exported into various document standards. In addition it is possible to establish a forward and backward reference between the textual requirement and actual source code.

PTC Integrity [138] is an application PLC system which offers an integrated Issue Tracking System (ITS) commonly used to track specifications. In addition it offers an interface to IBM Doors, which may be a common way of interacting with the requirements system.

Sparx Enterprise Architect [139] is a Unified Modeling Language (UML) editor which offers to write and maintain specifications in a graphical manner following the well known UML notations. Also, it is able to connect with IBM Doors and act as a graphical editor or exchange requirements.

The dominating testing method in RU is manual testing which represents

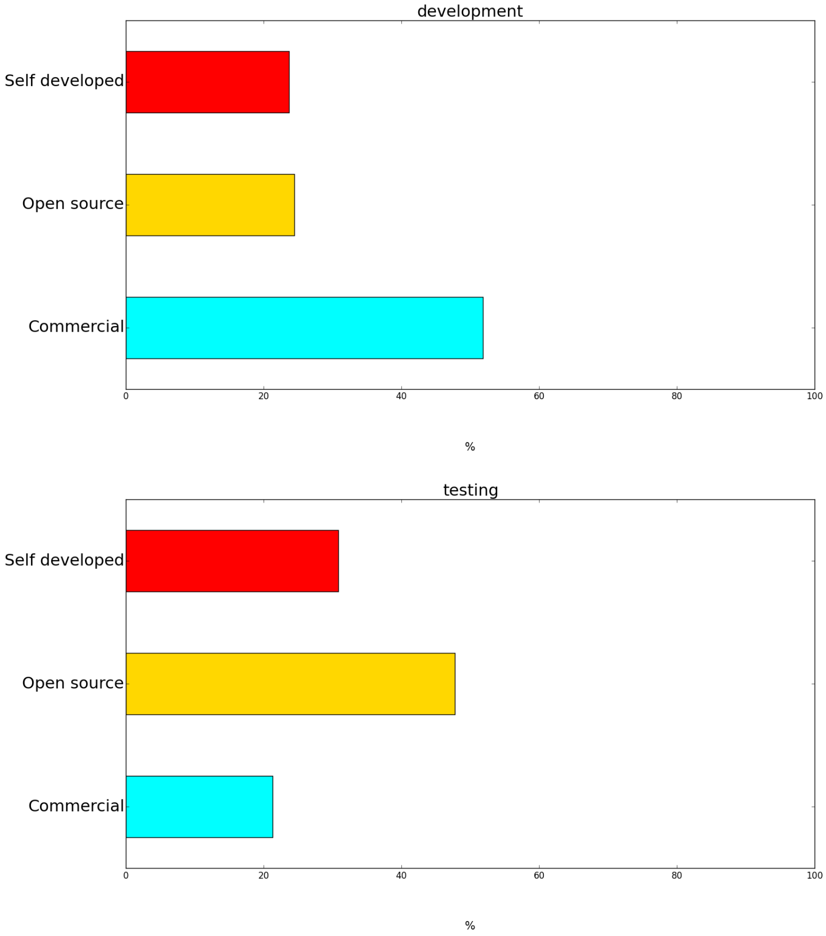


Figure 4.4.: Tool license in use as a result of a Survey by Altinger *et al.* [13]

Table 4.2.: test automation tools

name of tool	vendor	[%]	citation
Exact	Audi Electronics Venture GmbH (AEV)	7	[140]
Polyspace	the mathworks	7	[141]
EXAM	MicroNova	6	[142]
self developed		6	
TPT	PikeTec	5	[143]
CANoe	Vector Informatik	4	[144]
QA-C	QA Systems	4	[145]

the manual execution of TC according to Figure 4.5. This may be caused by the high number of functional changes and the innovative character of the department where less formal requirements are specified. In terms of SD and PD manual TC execution may refer to tests of the prototype car where it is necessary for a human driver to execute certain road test scenarios. Automated testing and test suite management are mainly used together with SiL and HiL tests, which are the most performed test stages see Figure 4.9. These results suit the popular[5] test automation tools listed in Table 4.2. EXact, Exam and TPT offer the automated execution of regression tests but still need manual TC specification. CANoe offers a scripting interface to automate test execution, Polyspace and QA-C are static code analysers.

Fitting data from Table 4.2 and Figure 4.5 the tool usage as presented in Figure 4.6 shows the majority upon test automation with a specific interest on code coverage by the TC within SD departments as this is a required performance ratio. More recently, static code analysis gained interest as a more industrial grade and easy to use tools became available *example given* Polyspace [141]. About a quarter of the cases code review is performed where Shull *et al.* [20] state to find on average 60% of the defects. On contrast RU seems to apply more mature methods as static code analysis is performed more often and even formal methods seem to be in use.

The main approach of testing is model based as Figure 4.7 shows. This may be caused by a high number of MiL tests already performed on the Matlab

[5] as a result of the surveys question upon the test automation tool in used

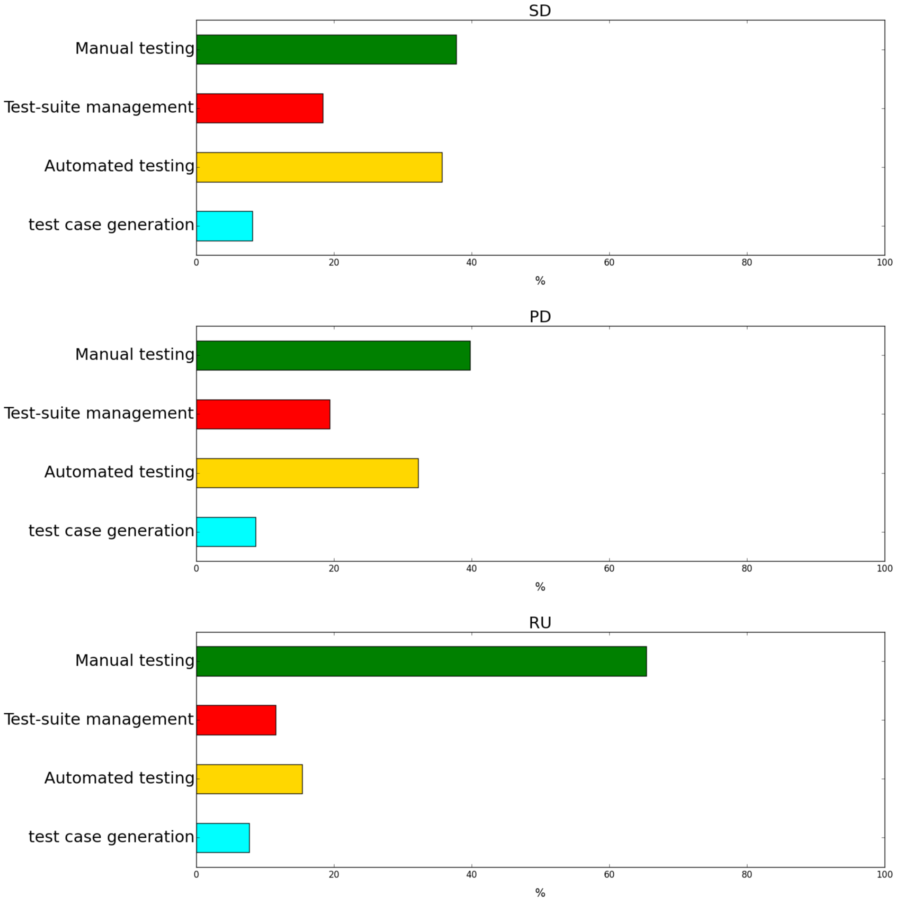


Figure 4.5.: Methods to automate testing as a result of a Survey by Altinger *et al.* [13]

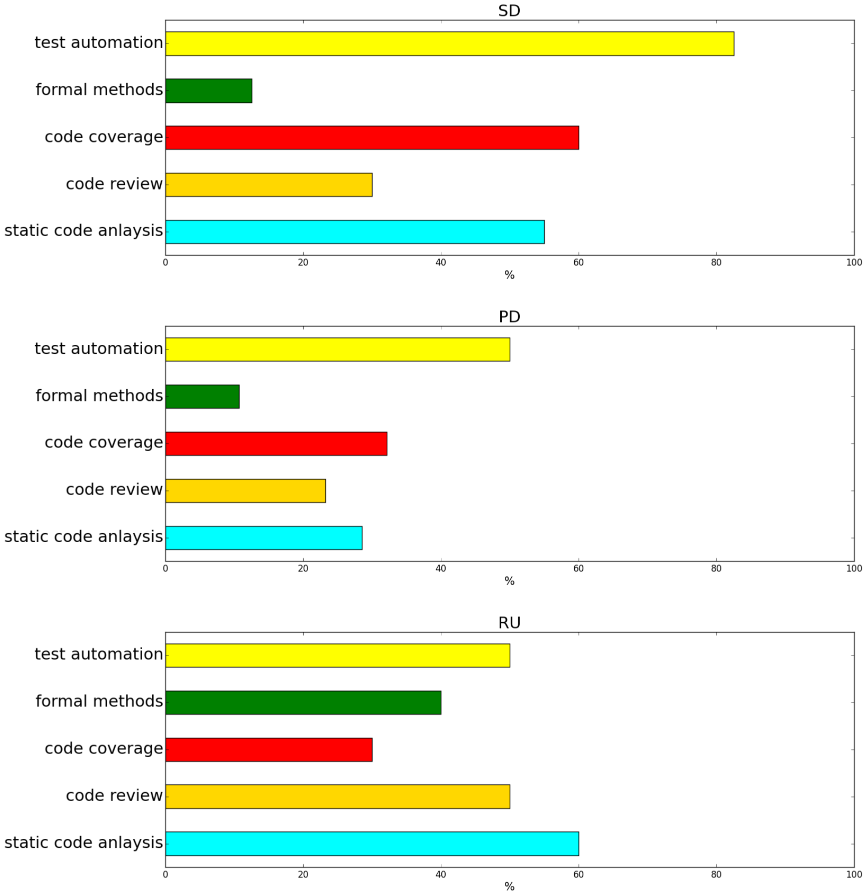


Figure 4.6.: Test tool usage as a result of a Survey by Altinger *et al.* [13]

Simulink models. It is worth mentioning that a high share of unit tests are performed across all three department domains. One might see an ECU as a unit and all tests *example given* HiL which target one single ECU (compare Figure 4.10) might be considered as unit testing. Modern testing approaches like fuzzy, mutation or random testing are hardly in use which could be caused by norm requirements *example given* ISO26262 [49] which forces tests to be deterministic.
There is a general idea that when designing a TC, "positive testing" denotes checking for the desired output which may be specified by the requirements or feeding valid input data, "negative testing" means specifying a test to use invalid input data and checking for a correct error statement. "Outside specification testing" is only possible if certain input value ranges are specified as this method tests for input values which are not defined. This may be something like if a function is specified to be available if the speed is below 12km/h a TC tests with a negative speed or something huge *example given* 3.000km/h, which is clearly outside the specifications of a passenger car. Figure 4.8 shows a roughly equally distributed arrangement among these three basic ideas. This means that automotive TC covers desired and none desired behaviour of specifications.
As described in Section 2.3 "in the loop" tests are specific to automotive software engineering. As visualized within Figure 4.9 HiL and SiL are the dominating test execution platforms at SD and PD. Noteworthy SD need to qualify the processors and ensure correct runtime even prior the desired ECU is not available, thus PiL account to a rather high share. Overall three domains "whole vehicle" testing is important, specially within RU where all functions need to be demonstrated running on an actual car. Simulation seems to gain attraction as it is the second highest ranked method in PD and RU seems to simulate every component prior to testing in the lab car. This may be the result as more complex functionality might require more time to Setup and prepare a prototype car or specific environmental conditions are hard to reproduce.
Testing software libraries seems to have an equal share among the SD,PD and RU, a clear difference is the high share of ECU testing at SD and whole vehicle testing at RU. This is in line with Figure 4.9 where RU performs a clearly higher share of tests in the car than on test beds.
As shown in Figure 2.4 SPICE and ISO 26262 [49]are the most dominant process Norms, AUTOSAR the most dominant implementation relevant

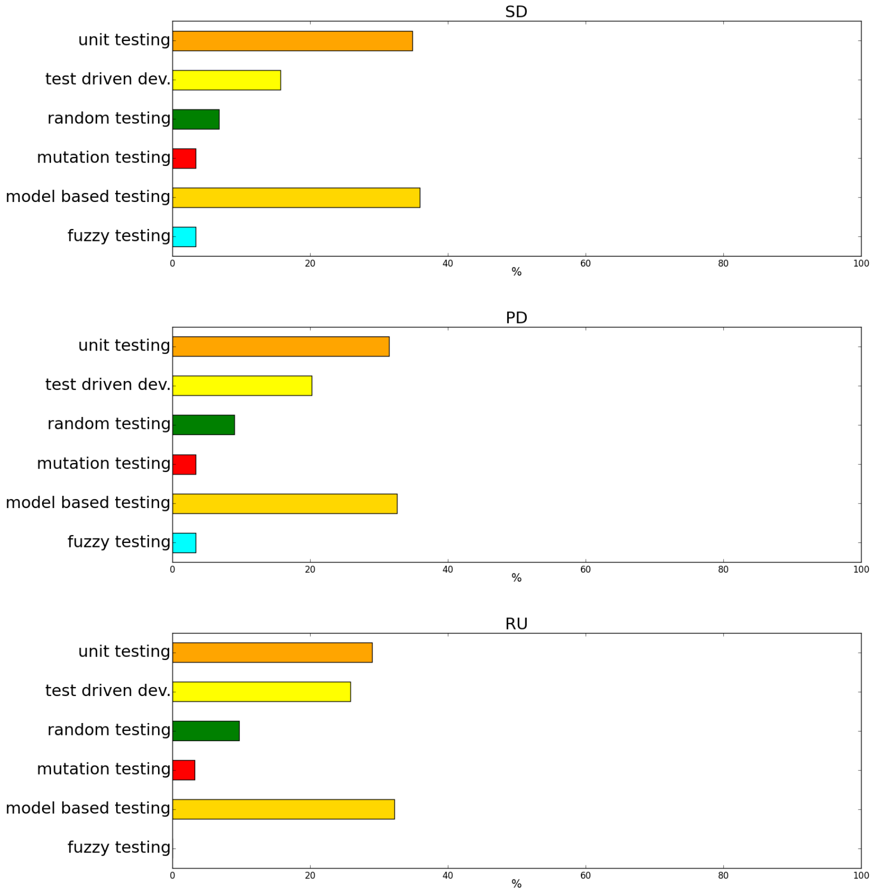


Figure 4.7.: Testing approaches as a result of a Survey by Altinger *et al.* [13]

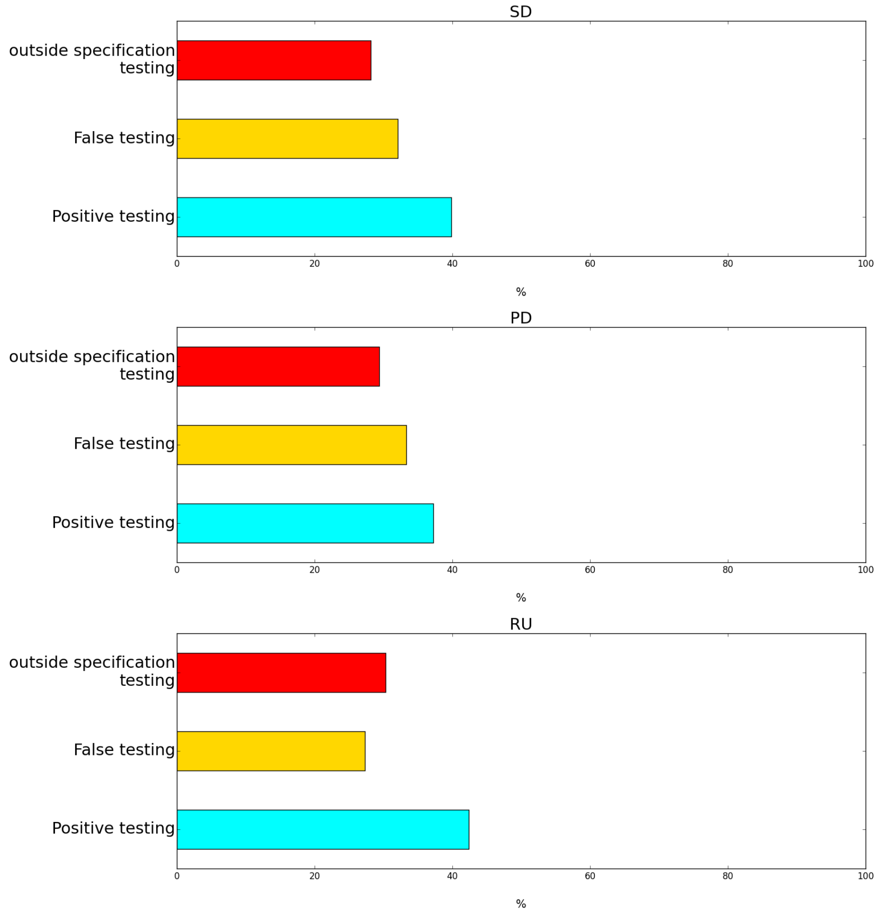


Figure 4.8.: General type of testing as a result of a Survey by Altinger *et al.* [13]

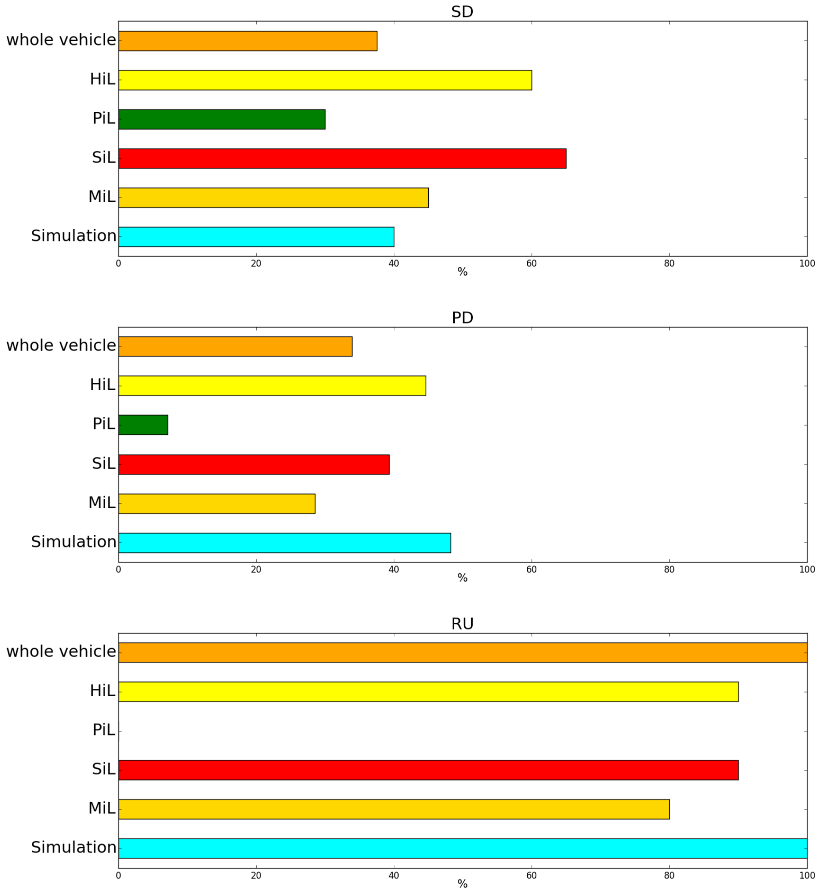


Figure 4.9.: Test time along the milestones as a result of a Survey by Altinger *et al.* [13]

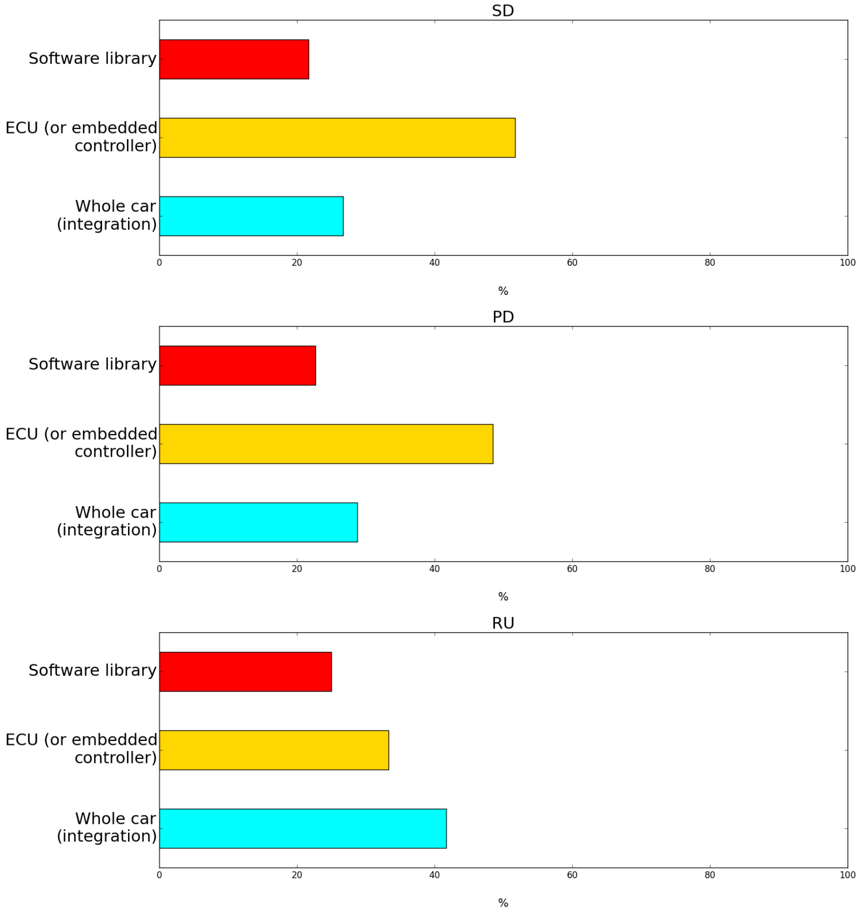


Figure 4.10.: Development target as a result of a Survey by Altinger *et al.* [13]

norm. SPICE defines the type of process to consider and the reports to exemplify, ISO 26262 [49] influences the way a TC needs be defined and the functionality it covers. In terms of TC definitions the AUTOSAR standard influences the way a TC can interact with the SUT, as AUTOSAR mainly defines interfaces and standard architectures to considers.

The majority of TC designers and execution engineers within the SD departments are from a third party which lead to the statement that testing is a dedicated job, compare Figure 4.11. RU and PD seem to highly share developer and tester. This fits together quite well, as the majority of tests are performed manually and the test's scope is mainly on the whole car (aka. full system) for those two department domains. This may be a result of the required test report effort being lower than in SD and the number of norms and laws to consider is even lower.

The final question has been to the participant's opinions concerning the usefulness of software testing, refer Figure 4.12. There is a clear statement upon the usefulness of testing (Q1) and the chance upon Return Of Investment (ROI). Q5 and Q3 are in line with Bock *et al.* [38]. New tool chains cost too much time as less than 20% think the time to design TC is less than implementing the actual software. Q4 can be seen as undecided as the median is more than 50% which can be interpreted as ”do not know” or ”no plans yet”.

4.1.3. Summary

Learning from Figure 4.12 the majority of survey participants thinks testing is an important work and the time invested is worth it. In general there are differences between the three department domains RU, PD and SD, the most obvious between SD and PD on one side and RU on the other side. Clearly SD need to consider laws and norms and spend more time to Setup unit and ”in the loop” tests for delimited functions blocks, see Figure 4.9 and 4.10. RU seems to apply a high amount of simulation which is equally ranked as whole vehicle tests, see Figure 4.9. Modern testing methods like fuzzy, random or mutation are rare, compare Figure 4.7, however, more tools using mature testing and analysing approaches are being applied, at

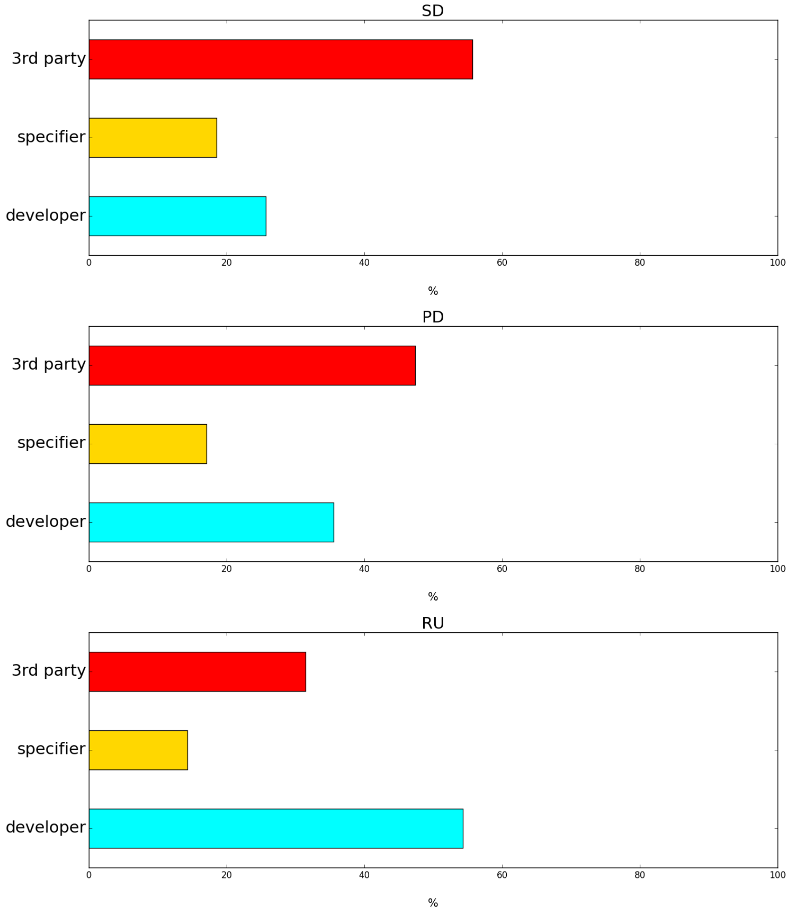


Figure 4.11.: Personnel to design TC as a result of a Survey by Altinger *et al.* [13]

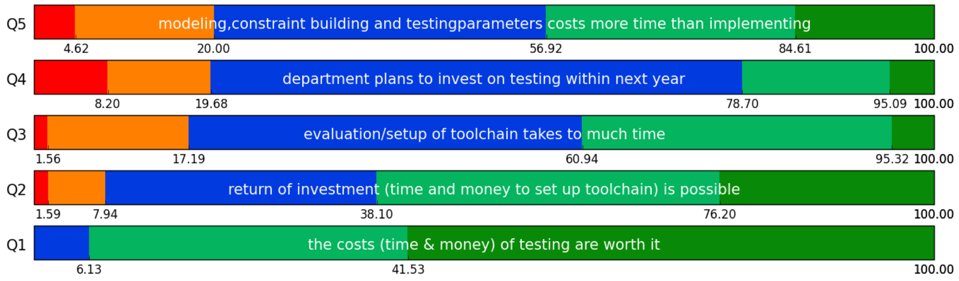


Figure 4.12.: Opinion about ecologically worthwhile of testing results from Survey by Altinger *et al.* [13]. Questions could be answered similar to school grades from 1 as total agree to 5 as do not agree at all. Starting with red on the left side it represents a full disagree and ending with dark green on the right side as a full agree.

least at RU see Figure 4.6. The majority of TC is designed and even a high share of them is executed manually, see Figure 4.5. This matches the top five list of test automation tools according to the survey participants which can be extracted from Table 4.2. The personnel to write specifications, design TC and implement the software is dedicated and different, see Figure 4.11, at least for SD departments.

Matlab Simulink is the most dominating tool to develop software within the automotive domain. Dedicated personnel is responsible for writing specifications, developing the code and defining the TC. A free TC budget is available and spent according to an engineer's experience.

4.2. Development Workflow

As presented by Broy *et al.* [31], Altinger *et al.* [13] and Bock *et al.* [38] Matlab Simulink is among the most common development systems within the automotive software industry. Altinger *et al.* [23] and Kiffe and Bock [146] present a workflow using model driven approaches by Matlab to develop embedded automotive software systems. Figure 4.13 visualizes the involved tool chains as used by AEV.

Requirements are composed using IBM Rational Doors [137], a database system to manage structured requirements with the ability to export and link them to various documents including Matlab Simulink models and source code. This feature enables the workflow to establish a direct link between graphs within the model and the dedicated textual requirement, upon request in full detail. These requirements will be used by either the developer when designing the model or the test engineer when writing the test specification.

Models are designed by dedicated model engineers using Matlab Simulink [41] and indexed using a Source Control Management (SCM) system, in this case PTC Integrity [138]. This SCM system offers services comparable to SVN, but go further. A core feature enables the project manager to set commit policies like forcing documentation or a link between the new revision and an ITS ticket. As well as PTC Integrity offers an integrated ITS which is used to report and track bugs but even to plan releases and feature commits.

Embedded systems still need a native code which can be compiled from C code using a target platform compiler. This required C code is to be automatically generated using dSpace TargetLink [42] and operated by a dedicated code generation engineer. Such a code generator uses customizable patterns to translate model blocks and elements into source code.

To ensure coding standard conformity according to MISRA a static code analyser like Mathworks Polyspace [141] is used. Prior to the model level conformity checks to ensure compatibility to Mathworks Automotive Advisory Board (MAAB) modelling guidelines [147] are performed using Model Engineering Solutions MXAM [148]. The SCM system can be configured with policies to enforce coding standard checks before accepting a commit.

Requirements may be updated or changed during development if they are misleading or developers and tests discover missing conditions or unintended behaviour. Models may be updated more frequently than generated code, thus every source code commit has a corresponding model commit but not vice versa.

Kiffe and Bock [146] describe the EntwicklungsProzess Verbesserung, german: Development Process Improvement (EnProVe) collection as a workflow process and a framework to incorporate various development tools. This process also performs the tool qualification as required by ISO26262 [49] which is a prerequisite for every tool allowed during automotive software development for safety critical systems, obversely developers are not free to choose whatever tools suits them.

A recent study by Bock *et al.* [38] upon tool's usage and the developer's association with processes evidences 100% usage of the V-development process, where Kiffe and Bock [146] present a suiting EnProVe tool for every stage.

After the initial start of a project, the tool suite isn't changed until the final release at SOP of a car, thus there is not even an update of compilers, checkers, IDE, *etc.* on minor or major releases. Even configuration settings like the code generator templates or compiler options are not changed after an initial tuning phase. This realizes static development settings and ensures reproducibility during tests as required by various standards and norms like ISO 26262 [49].

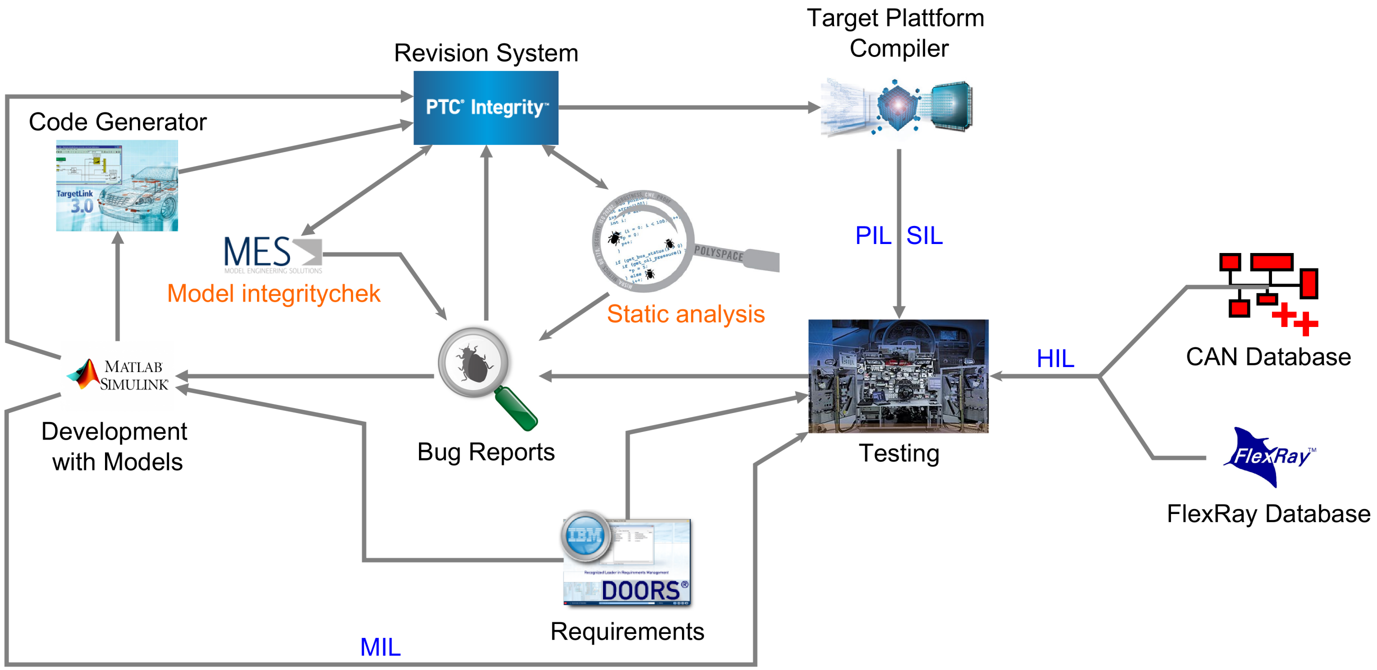


Figure 4.13.: Software development Workflow with incorporated tools, adopted from Altinger *et al.* [24]

5. Analysis of real world Automotive Software Projects

The NASA metric data program [133] (NASAMDP) [133] and the PROMISE repository [131] host a wide range of datasets, as outlined in Section 3.4, however none of them originated within automotive industry or developed using model based techniques. This chapter describes a dataset aggregated from three automotive embedded software projects including analysis of the data and its ability to be used for SFP. The dataset is publicly available for download at [149]. Section 5.1 introduces the origin of the data, Section 5.2 explains the datasets creation process and the available software metrics, Section 5.3 presents correlation analysis of the data, Section 5.4 presents the bugs distribution and Section 5.5 gives an overview to the types of errors discovered during development and their influence upon the metric data.

5.1. Unique Dataset

This section presents a unique dataset from the automotive domain offering source code and model metric data along with bug information for three real world automotive software projects developed in-house at AEV. Following the definition by Radjenović *et al.* [125] the dataset is midium-sized containing metrics on 59.104 LOC in total. To date this is the only public available data from the automotive domain. The presented content is based on Work by Altinger *et al.* [23] and Altinger *et al.* [26]. The advertised dataset is free of charge and can be downloaded following [149].

The code is developed using Matlab Simulink [41] to design the algorithms and the overall model, state machines and decision trees are realized using Stateflow [150]. The actual source code is generated using dSpace TargetLink [42] and has no manual code changes, thus the code is a one on one translation from the model. The complete development process is described within Section 4.2 and illustrated in Figure 4.13.

All projects follow the MISRA 2004 coding standard [44] and the MAAB model guidelines [147]. Both set of rules are checked with tool support and their correctness is enforced by policies on the SCM system. The developer is not able to commit his work as long as the tools report coding or model standard violations. Thus the code it self represents a very high development quality. In contrast, public available datasets, see Section 3.4, are mainly aggregated from open source software. Following the analysis by Stamelos *et al.* [151] where not all projects conform to such high standards, the presented dataset is unique in terms of coding guidelines. For confidentiality reasons, the projects are named A,K and L and no further information concerning the actual functionality or the car where they have been released first can be given. The software is out on the roads and there is no single report upon misfunction or failure over a period of many years. Thus one can argue all relevant errors have been found and the bug history is complete.

Table 5.1.: Project description and data overview: 'number of files' represents the unique number of files, 'committed files' represents the sum of all commits over all files, 'error prone files' represents the number of files marked as 'real code bug' over all commits

	num. Requirements	num. Sub-projects	LOC	num. Testcases	num. Authors	num. src. files	num. mdl. files	commited files	error prone files	Software Type	AUTOSAR	Safety functionality
Project A	304	13	12.465	185	4	45	26	1.908	77	logic, timing dependent behaviour	yes	no
Project K	900	24	36.526	695	5	53	48	2.515	112	mainly logic operations and branching	yes	yes
Project L	600	8	10.113	680	3	20	47	2.891	61	logic, timing dependent behaviour	yes	yes

Table 5.2.: Projects fault distribution overview, the number of commits is reported

	commits		fault		
Project	feature	bugfix	free	buggy	bug rate
A	1894	14	1820	88	4,61%
K	2448	71	2144	375	14,89%
L	2880	12	2816	76	2,63%

Table 5.1 presents an overview of the project's characteristics, listing the size in terms of LOC and the number of requirements. Each project consists of a main module and a number of sub modules. These are developed and tested independently and integrated into the full system. All three projects are realized as software components defined within the AUTOSAR standard. Consequently this indicates there are no low level functions as device access drivers or micro code maintaining a processor's interruption or similar. The development team was constant over the full project's time as well as the development tools. K is the most mature project by terms of the number of requirements, team size, LOC and number of sub projects. L and K are rated as safety relevant classified as ASIL B which raises the effort required to test the software as one can see in the number of test cases.
Peleska *et al.* [37] did release a comparable model for a car's turn indicator. This visualized the root causing a high number of boolean conditions, where the presented model is otherwise simpler than the analysed project. One can see a frequent boolean condition ensuring no false activation of the turn indicator, even the overall system description is rather easy "if the driver activates the related indicator lever, flash the front and rear turn light".

Table 5.2 lists the buggy and fault free commits of the three projects illustrating the low fault density. Only 0,7 - 3% of the commits were explicit bug fixing, others are feature commits. As identified by the SZZ Algorithm 2,61% - 14,89% of all commits are containing a bug. The only publicly available and comparable[1] data is the NASAMDP project PC1 and CM1 contain 7% and 10% of faulty commits. Other datasets, *example given* Eclipse by Zimmer-

[1] in terms of programming language, restrictive development settings and safety relevance

Table 5.3.: Operator overview: Preprocessor statements

Preprocessor statements					
A		**K**		**L**	
%	Operator	%	Operator	%	Operator
16,86%	#define	20,56%	#define	16,91%	#define
16,86%	#ifndef	14,65%	#ifndef	16,91%	#ifndef
66,28%	#include	64,79%	#include	66,18%	#include

man *et al.* [93], originated from development history with more than 30% of defective files. This dataset suffers from a class distribution around 2:100 - 14:100, using the definition by Weiss [126] the presented dataset represents a mid-class imbalanced two class problem. Machine learning algorithms are known to have problems on such datasets, compare Khoshgoftaar *et al.* [116].

An analysis of the used operators can be extracted from Tables 5.3 to 5.8. Due to the project's nature, the source code consists mainly of logic operators, compare Table 5.1. This is a good indicator upon the types of software errors which are possible by the used programming language features, see Section 5.5. Bearing in mind the huge number of requirements, where the majority is related to preventing misuse cases or defining value condition ranges, and the structure which can be seen within the turn indicator model by Peleska *et al.* [37], causes a high number of conditional operators to be in the code too.

The dataset offers typical software metrics as used in SFP literature like Graves *et al.* [67], Ostrand *et al.* [78] or Menzies [88]. Amongst others *volume metrics* such as LOC, Halstead (Volume, Effort and Difficulty) and a number of functions. Furthermore *code structure metric* as CC, *change metric* as code chrun (LOC add and LOC remove), *file and commit history* as a number of commits and age of commits. But most importantly every commit is marked as either fault free or containing a bug. Table 5.9 presents the full set of source code metrics and Table 5.10 the full list of change metrics.

In Addition the dataset contains metrics on the Matlab Simulink Models

Table 5.4.: Operator overview: Logic and comparator

Logic and comparator					
A		**K**		**L**	
%	Operator	%	Operator	%	Operator
3,59%	!	12,39%	!	0,24%	!
2,12%	! =	2,00%	! =	6,49%	! =
2,74%	&	9,02%	&	3,49%	&
6,60%	&&	15,14%	&&	10,33%	&&
76,38%	=	46,17%	=	43,88%	=
1,69%	==	6,87%	==	19,78%	==
0,32%	|	2,38%	|	2,03%	|
2,90%	|	5,07%	|	5,43%	|
0,27%	>=	0,13%	>=	3,30%	>=
0,29%	<=	0,06%	<=	0,35%	<=
1,25%	<	0,41%	<	1,51%	<
1,80%	>	0,32%	>	1,72%	>
0,05%	∼	0,00%	∼	1,44%	∼
		0,05%	ˆ		

Table 5.5.: Operator overview: Variable change

Variable change					
A		**K**		**L**	
%	Operator	%	Operator	%	Operator
100,00%	>>	1,07%	>>	54,44%	>>
		85,59%	<<	45,56%	<<
		13,34%	++		

Table 5.6.: Operator overview: Flow control

Flow control					
A		**K**		**L**	
%	Operator	%	Operator	%	Operator
17,55%	break	3,40%	break	12,27%	break
14,12%	case	2,62%	case	9,04%	case
3,43%	default	0,79%	default	3,24%	default
24,91%	else	34,42%	else	24,73%	else
1,06%	for	0,70%	for		
35,31%	if	55,49%	if	47,46%	if
0,08%	return	1,53%	return		
3,43%	switch	0,79%	switch	3,26%	switch
0,12%	while				
0,00%	?	0,27%	?		

Table 5.7.: Operator overview: Datatypes

Datatypes					
A		**K**		**L**	
%	Operator	%	Operator	%	Operator
95,38%	static	95,90%	static	99,06%	static
4,62%	struct	0,29%	struct	0,94%	struct
		1,91%	unsigned		

Table 5.8.: Operator overview: Mathematical

Mathematical					
A		**K**		**L**	
%	Operator	%	Operator	%	Operator
23,37%	+	22,98%	+	88,17%	+
37,86%	-	4,34%	-	11,30%	-
		4,44%	%		
18,49%	*	55,19%	*	0,52%	*
20,28%	/	13,05%	/		

which are described by Altinger *et al.* [23] in detail. As they are not part of this Thesis work the reader is redirected to this publication.

Table 5.9.: table containing source code metrics

fieldname	comment	tool
filename	filename	python
project_name	subproject name	python
hash	md5 file hash	python
lm_LOC	LOC	[152]
lm_SLOCP	Physical Executable LOC	[152]
lm_SLOCL	Logical Executable LOC	[152]
lm_MVG	CC	[153]
lm_BLOC	Blank LOC	[152]
lm_CNSLOC	Code and Comment LOC	[152]
lm_CLOC	Comment Only LOC	[152]
lm_CWORD	Commentary Word	[152]
lm_HCLOC	Header Comment LOC	[152]
lm_HCWORD	Header Commentary Words	[152]
h_N1	number of total operators	[153]
h_N2	number of total operands	[153]
h_n_1	number of unique operators	[153]
h_n_2	number of unique operands	[153]
h_V	Halsted volumen	[153]
h_D	Halsted difficulty	[153]
h_E	Halsted effort	[153]
last_modified_date	TargetLink generation time	python
src_mdl_file_hashes	hash from mdl	python

Table 5.10.: table containing change metrics

fieldname	comment
filename	filename
project_name o	subproject name
author	authors name [surename.firstname]
filetype	type of file (header or source)
project_rev	PTC Integrity revision number
date	date of entry
svnrev	SVN revision number
num_bugs_trace	number of buggy lines
pot_bugs	number of 'bug', 'error' in commit message
num_bugs	number of fixed bugs

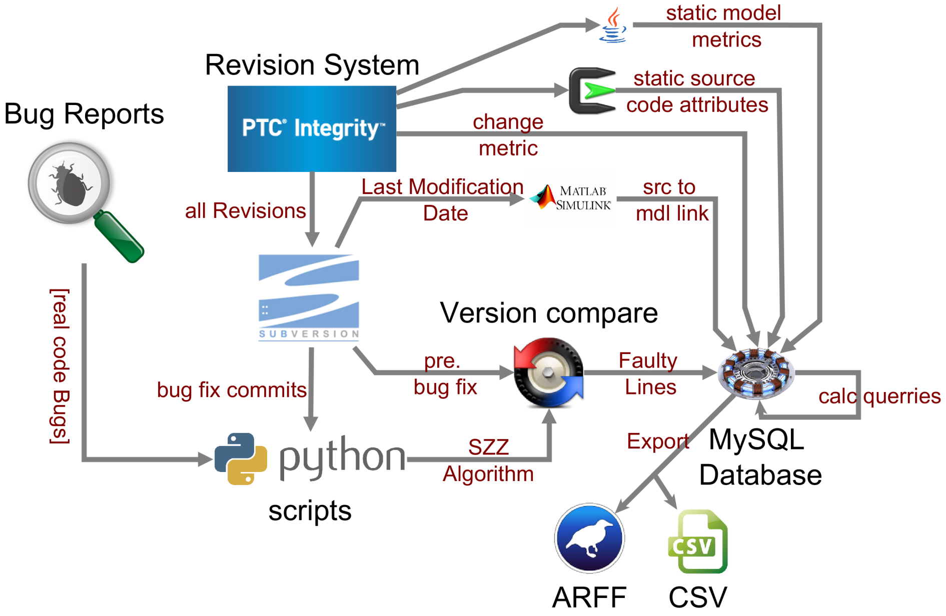


Figure 5.1.: Workflow to create the Datasets, adopted from Altinger *et al.* [23]

5.2. Creation of the Dataset

The dataset is created using a number of python scripts to automate data collection, the presented content is based on Work by Altinger *et al.* [23]. An overall visualization of the datasets creation process can be extracted from Figure 5.1. The following passage explains the single steps and tools in use.

The core script uses an API supplied by the SCM system PTC Integrity [138]. For every single project all committed revisions are checked and re-committed into an SVN repository maintaining a direct link between PTC Integrity and SVN revision number without effecting any file and commit history. This is a required step as PTC Integrity does not offer a command similar to ”svn annotate” which reports the revision number since a selected line exists without any changes. Using the same API another script queries the PTC Integrities ITS to gather all tickets classified as a ”trouble ticket” with a sub category ”real code bug”. Every ticket is associated with a revision

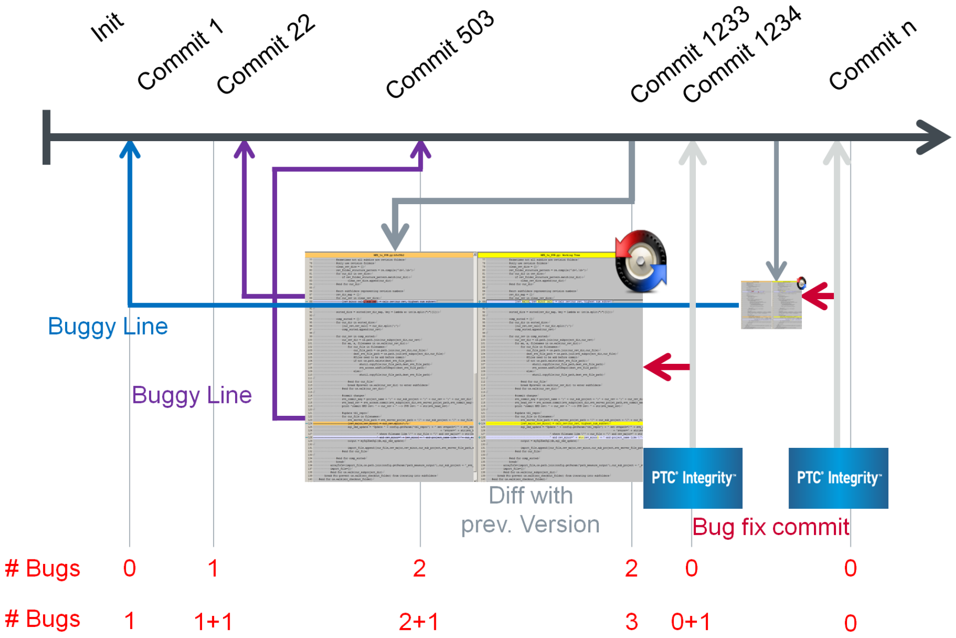


Figure 5.2.: Visualization of the SZZ Algorithm by Kim *et al.* [83] applied to the dataset

where the bug is fixed. Figure 5.2 shows the next stage, using Step one to four from the SZZ-Algorithm by Kim *et al.* [83]. This determines which revision contains a bug and mark them. Scooter Software Beyond Compare [154] is used to generate a file diff between the bug fixed revision and its predecessor unveiling the buggy lines and the required code changes to eliminate them. Beyond Compare is chosen among its ability to report changed lines as eXtensible Markup Language (XML) and apply regular expressions (regex) filters. The metric data is calculated using public available open source tools LocMetric [152] and cMetric [153], see Table 5.9 for an exact mapping between metric and tool. All meta data as file name, authors, *etc.* is collected using python scripts and regex queries. All data is stored within a MySQL [155] database to enable further data processing. Code chrun and file & commit history is calculated using Structured Query Language (SQL). Further python scripts are used to generated export Formats as WEKA's ARFF and CSV.

The original release of the dataset by Altinger *et al.* [23] contains descending sorted chronological metric and bug data. This is corrected by Altinger *et al.* [25] when using ascending sorted data. In contrast to Altinger *et al.* [24] Altinger *et al.* [25] achieves significantly higher SFP results.

5.3. Metric data Analysis

To determine which software metrics to use and ranking their influence upon fault prediction correlation analysis is performed. The presented content is based on Work by Altinger *et al.* [23].

Demšar [156] argues bug data is non Gaussian therefore it is required to use a non parametric correlation tests. A Shapiro-Wilk [157] test performed on all three projects and all metric data resulted in p-values below 0.05, which proves non-Gaussian distribution of the input data. Based on these insights one need to use none parametric tests when analysing the data. Tomaszewski *et al.* [86], Turhan *et al.* [110], Menzies *et al.* [114] and D'Ambros *et al.* [118] suggest using the Mann-Whitney U [158] test. Olague *et al.* [159] uses Kendall τ [160], Tomaszewski and Damm [86], Mende and Koschke [106], D'Ambros *et al.* [113] apply Pearson [161] or Spearman rank [162] as both a known to deliver comparable values. Following these suggestions Tables 5.11, 5.13 and 6.11 present the obtained results which were calculated using pythons scipy[2] library.

Table 5.11 presents the Kendall τ and Pearson measure. Clearly one can see a high correlation between static volume metrics as LOC and Halstead (volume, difficulty and effort) and even code structure metrics as CC. This behaviour is reported multiple times in literature by others. Overall there is a remarkably weak correlation around zero to the committing author. This may be caused by the fact that the code is not written by hand and the code generator derives the code in exactly the same way for every author. Furthermore the MISRA coding standard and MAAB model guidelines reduce the variance in building up models alongside the authors, so there is no room for a "personal coding style". Code chrun (LOC add and LOC remove) shows a weak correlation tendency to which might be caused by

[2] https://www.scipy.org/

an average of 25 LOC added and 15 LOC removed per commit over the full dataset. In total there is a weak correlation to bugs by all measured metrics. This is similar to values reported from the NASAMDP datasets, *example given* Pearson correlation between LOC and Bug on the PC1 project is weak too, compare Table 5.12.

As suggested by literature work (*example given* Menzies *et al.* [114]) the Mann-Whitney U test is applied to the test set as well as the Spearman correlation, see Table 5.13. The results are comparable to Table 5.11 and Table 5.12 with a weak correlation between any metric and bug data. Within the reported metrics Halstead and LOC as well as CC and "number of functions" are highly correlated.

In addition to the analysis from Table 5.11 and 5.13 where there is no significant correlation between the metrics and bug data Table 5.14 presents an analysis using InfoGain to rank the metrics influence. The most dominating are LOC, Halstead Volume (HV) and HE which are known to be correlated. Code chrun and author information carry a low information gain.

Based on the analysis presented in Table 5.11 and 5.13 the dataset shows similar behaviour than others reported in literature, thus it is expected to enable SFP studies.

Table 5.11.: Correlation analysis on selected metric data from project K, comparing Pearson against Kendall τ, strong correlations are marked in bold.

		Kendall τ											
		author	LOC	CC	Hv	Hd	He	LOC add	LOC remove	num functions	commit age	number commits	bug
Pearson	author		0,01	0,00	0,02	0,00	0,01	0,05	0,04	-0,01	-0,14	-0,12	0,05
	LOC	0,07		**0,78**	**0,91**	**0,85**	**0,92**	0,39	0,39	**0,71**	0,05	0,09	0,38
	CC	0,05	**0,92**		**0,77**	**0,74**	**0,78**	0,41	0,42	**0,81**	0,05	0,09	0,40
	Hv	0,06	**0,97**	**0,86**		**0,84**	**0,94**	0,37	0,36	**0,70**	0,04	0,10	0,37
	Hd	0,05	**0,75**	**0,87**	**0,70**		**0,90**	0,38	0,37	**0,70**	0,04	0,07	0,36
	He	0,07	**0,93**	**0,89**	**0,95**	**0,76**		0,38	0,37	**0,70**	0,04	0,09	0,37
	LOC add	0,08	0,37	0,38	0,33	0,37	0,37		**0,81**	0,44	-0,11	-0,11	0,23
	LOC remove	0,07	0,41	0,41	0,36	0,38	0,40	**0,74**		0,44	-0,02	-0,03	0,22
	num functions	0,01	0,55	**0,71**	0,48	**0,79**	0,53	0,26	0,29		0,04	0,04	0,43
	commit age	-0,16	0,06	0,10	0,07	0,05	0,08	-0,03	0,03	0,08		**0,74**	-0,10
	number commits	-0,13	0,24	0,26	0,26	0,14	0,27	-0,01	0,07	0,14	**0,83**		-0,07
	bug	0,05	0,52	0,49	0,47	0,49	0,48	0,26	0,24	0,35	-0,14	-0,09	

Table 5.12.: Correlation analysis on selected metric data from NASAMDP PC1 project, comparing Pearson against Kendall τ. Strong correlations are marked in bold. Some values are not reported due to not reported metric data in the PC1 dataset. The tables layout is identical to Table 5.11 for better comparison.

		Kendall τ											
		author	LOC	CC	Hv	Hd	He	LOC add	LOC remove	num functions	commit age	number commits	bug
Pearson	author												
	LOC			0,65	**0,84**	0,66	**0,80**						0,15
	CC		**0,86**		0,65	**0,70**	**0,70**						0,11
	Hv		**0,94**	**0,86**		**0,73**	**0,89**						0,14
	Hd		0,66	**0,71**	**0,73**		**0,83**						0,08
	He		0,64	0,65	**0,80**	**0,71**							0,12
	LOC add												
	LOC remove												
	num functions												
	commit age												
	number commits												
	bug		0,27	0,16	0,23	0,09	0,12						

Table 5.13.: Correlation analysis on selected metric data from project K, comparing Mann-Whitney U against Spearman. Mann-Whitney U is reported as relative value to its maximum at 300.000 due to readability. Strong correlations are marked in bold.

		Spearman											
		author	LOC	CC	Hv	Hd	He	LOC add	LOC remove	num functions	commit age	number commits	bug
Mann-Whitney U	author		0,01	0,00	0,02	0,00	0,02	0,06	0,05	-0,01	-0,19	-0,16	0,05
	LOC	0,00		**0,90**	**0,98**	**0,96**	**0,99**	0,51	0,50	**0,86**	0,07	0,13	0,46
	CC	**0,92**	0,22		**0,89**	**0,88**	**0,89**	0,50	0,50	**0,89**	0,07	0,12	0,45
	Hv	0,00	0,52	0,08		**0,96**	**0,99**	0,49	0,48	**0,84**	0,06	0,14	0,45
	Hd	0,25	0,40	0,60	0,12		**0,98**	0,49	0,48	**0,84**	0,06	0,10	0,43
	He	0,00	0,31	0,01	0,64	0,02		0,50	0,49	**0,84**	0,06	0,12	0,45
	LOC add	**0,97**	0,24	**0,77**	0,12	0,55	0,06		**0,84**	0,50	-0,15	-0,15	0,26
	LOC remove	**0,86**	0,18	**0,83**	0,09	0,46	0,04	**1,00**		0,50	-0,03	-0,03	0,24
	num functions	0,09	0,00	**0,87**	0,00	0,01	0,00	0,30	0,35		0,05	0,05	0,44
	commit age	0,08	**0,78**	0,11	**0,82**	0,17	0,49	0,20	0,16	0,04		**0,90**	-0,13
	number commits	0,15	0,54	0,49	0,12	**0,85**	0,02	0,48	0,39	0,05	0,20		-0,08
	bug	0,02	0,00	0,69	0,00	0,00	0,00	0,11	0,13	**0,73**	0,03	0,03	

Table 5.14.: Metric ranked according to their Infogain

Project A		Project K		Project L	
InfoGain	metric	InfoGain	metric	InfoGain	metric
0,2	**Hd**	0,166	**Hv**	0,133	**LOC**
0,182	**LOC**	0,135	**CC**	0,133	**CC**
0,178	**HE**	0,135	**Hd**	0,132	**HE**
0,165	**Hv**	0,126	**HE**	0,131	**Hv**
0,071	CC	0,123	**LOC**	0,111	**Hd**
0,033	commit_age	0,054	nfunctions	0,099	nfunctions
0,032	num_commits	0,035	commit_age	0,03	num_commits
0,025	nfunctions	0,024	loc_add	0,024	loc_remove
0,013	author	0,024	loc_remove	0,024	loc_add
0	loc_remove	0,009	num_commits	0,016	commit_age
0	loc_add	0	author	0	author

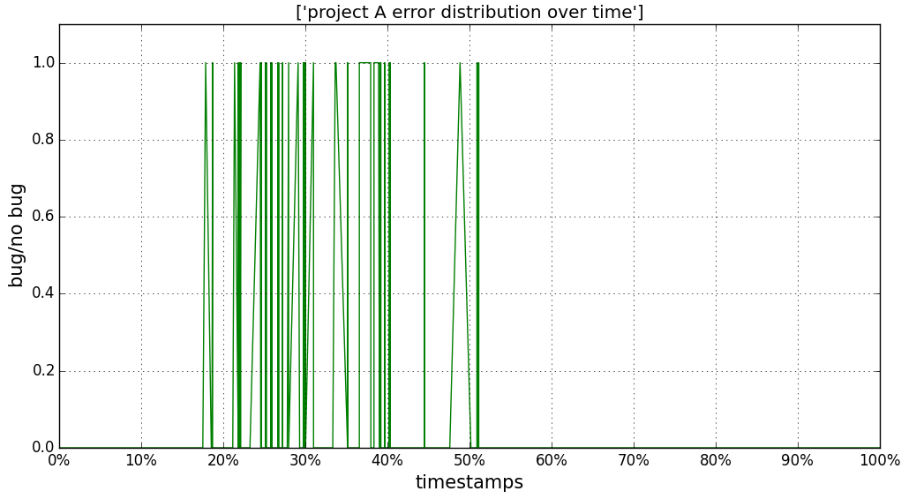


Figure 5.3.: Bug distribution within project A, 0 represents bug free, 1 is a faulty commit

There is no clear correlation between the recorded software metrics and the number of bugs occurring. Furthermore there is little correlation between the dedicated metric categories (static, code chrun and authors).

5.4. Bug Distribution

The dataset was created in chronological order, representing the time of commit. This causes non equally distributed bugs in all three projects, see Figures 5.3, 5.4 and 5.5. From the development history a first release milestone exists after one and a half years when the first version is delivered to the HiL test department.

Project A's first release milestone is after around 50% of the commits, so analysing Figure 5.3 one can see the majority of bugs is already fixed at the first release.
Project K's first release milestone is after around 40% of the commits, so analysing Figure 5.4 one can see the majority of bugs is discovered and

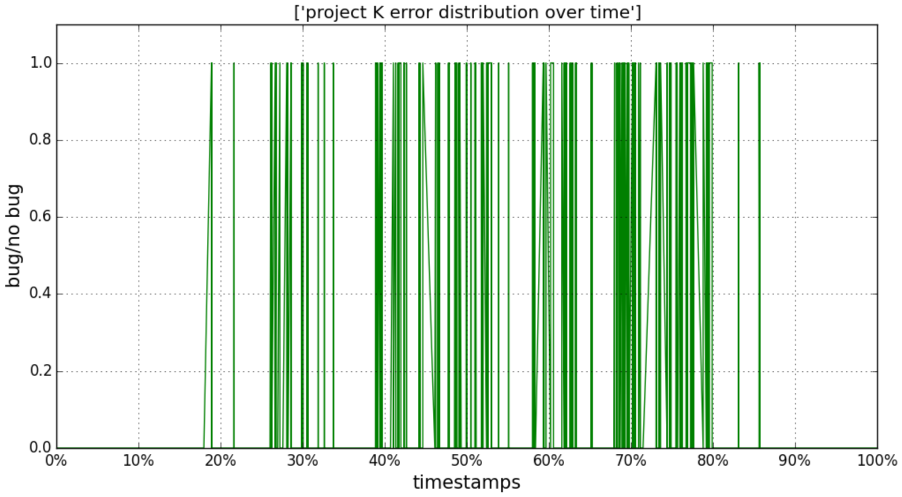


Figure 5.4.: Bug distribution within project K, 0 represents bug free, 1 is a faulty commit

fixed during testing phase.
Project L's first release milestone is from around 50% of the commits, so analysing Figure 5.5 one can see the equal distributed bugs prior to and after the first release.

5.5. Bug Analysis and Effects upon Preventive Measurements

Section 5.1 presents a dataset which is described by its software metrics, this section analysis the operators occurring within the source. The presented content is based on Work by Altinger *et al.* [26].

In total the dataset contains nine different categories of bugs, occurring in 483 error prone files. Project K, being the biggest consisting of the longest commit history, contributes the majority of faults. Table 5.16 lists all the bugs which occurred within the three projects. Due to the nature of the projects, see Table 5.1, the majority of bugs is boolean logic related, thus

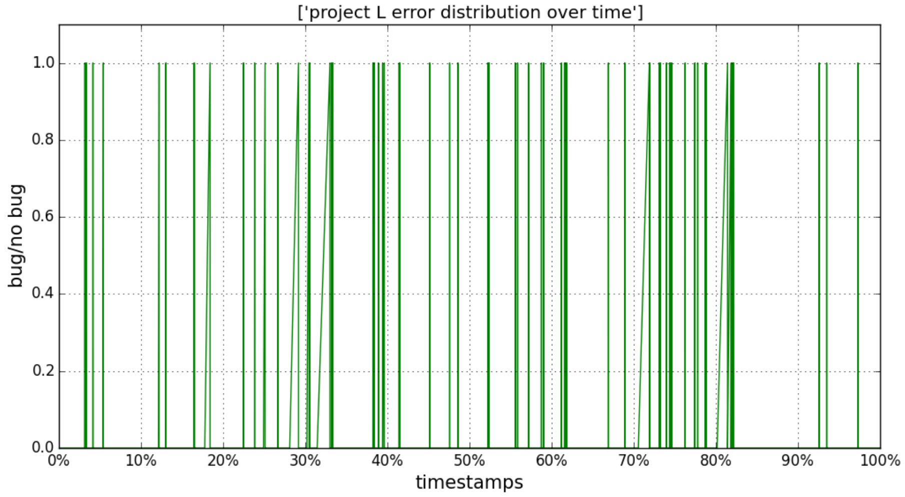


Figure 5.5.: Bug distribution within project L, 0 represents bug free, 1 is a faulty commit

fixing them contributed changed arithmetic or added boolean conditions. The majority of these changes does not effect any metric value, thus no bug causes the directly related change of a metric value. This is consistent with a comprehensive study by Fenton *et al.* [163] where the author states that bugs are not necessarily represented by software metrics. This may be a reason for the low correlation between bugs and metric values as presented in Table 5.11 and 5.13.

Further investigations unveil all condition statements ranging between one and eleven (mostly) boolean operands. 75% of the conditions affected by bugs have one to four operands with no tendency to one group, see Table 5.5. Hence there is no observable tendency of being buggy dependant on the number of operands.

As outlined in Section 4.2 the source code follows strict enforced MISRA guidelines which result in a limited feature set of the C language. Thus no pointers, no dynamic memory and no pointer operations are in use. Still the list of used operators, see Tables 5.3 to 5.8, would enable a broad range of possible error classes *example given while* and *for* loop bound errors, flow

Table 5.15.: Number of Boolean Operands from Observed Buggy Condition and their Occurrence Distribution

#operands	%
1	34,92
2	6,35
3	15,87
4	17,46
5	1,59
6	3,17
7	3,17
8	7,94
9	4,76
10	1,59
11	3,17

control, wrong value assignment.

Further there need to be more than removed language features to prevent multiple bug categories. The following rules are identified to guard typical errors:

- Rule *13.5*: "The three expressions of a *for* statement shall be concerned only with loop control".
- Rule *13.6*: "Numeric variables being used within a for loop *for* iteration counting should not be modified in the body of the loop".

Further rules like 14.8, 15.1, 15.2, 15.3 and 15.4 define the structure of a *switch/case* statement. This performs to prevents *example given* a missing break.
MAAB [147] guidelines force every Simulink block to enumerate its input signal and generate a case port for every possible value. Another guideline forces to connect every port, which guides a developer to condition every case.

The following MISRA rules prevent memory issues by forbidding dynamic memory:

- Rule *20.4*: "Dynamic heap memory allocation shall not be used"

The only known comparable study was performed by Wagner *et al.* [81] on industry OO software from the Telecom sector written in Java.

The existing analysed code development guidelines successfully prevent specific categories of bugs which could possibly occur, based on the utilized operators.

There is no clear correlation between the number of boolean conditions within a statement and the probability for a bug to occur.

Table 5.16.: Ranked Error Types with Explanation to all Three Projects. CC - see equation 3.1, N1 the Number of Total Operants, N2 the Number of Operators, n1 the Number of Unique Operators, n2 the Number of Unique Operants - compare 3.2.1. *n* Demands no Influence to Metric Value, *y* Demands an Influence, *w* Symbols a Weak Influence. Note, not all Error types occur in all three projects.

%	type	explanation	Project	LOC	CC	N1	N2	n1	n2
25,22	boolean logic error	miscalculations with boolean conditions	K,L	n	n	n	n	n	n
17,39	missing condition	missing variable within condition	K,L	n	w	y	y	w	w
15,65	wrong variable assignment	calculation result assigned to wrong variable	K	n	n	n	n	n	n
12,17	casting error	variable type declarations/cast	K,L	n	n	n	n	n	n
10,43	condition error	wrong path conditions calculated	K	n	n	n	n	n	n
9,57	mistmatch boolean, bitwise logic	e.g. < instead of <<	A,K	n	n	w	w	w	w
5,22	fixed Value error	wrong initial fixed values	K	n	n	n	n	n	n
3,48	calculation error	miscalculations with numeric values	K	n	n	n	n	n	n
0,87	wrong signal send	wrong output values	K	n	n	n	n	n	n

6. Fault prediction and Analysis upon Cross Project Prediction

Software Fault Prediction is a young research area with promising potential to advise spending of free TC budget across a software project. In general this method uses historic data to estimate a probability on an actual commit upon containing a bug or being fault free. The accuracy of the prediction strongly depends on the quality of the measurement data.

Rakesh *et al.* [164] analyse the automotive software PLC, stating SFP is helpful and applicable to the domain. They suggest multiple points in time where to apply SFP and which metric can be acquired. In their opinion SFP using software metric data is available during ”functional development” and ”integration” & ”testing” phase, prior to these phases they suggest the usage of expert opinions and transfering knowledge about common faults from similar projects.

Section 6.1 reports work on predicting fault within the same project as the software metrics, the actual measurement data, is acquired. To overcome deficits if the project contains too few faults Section 6.2 presents work on handling these imbalanced class problems. As there are projects with a short or no history at all, Section 6.3 presents results from reusing a trained predictor on another project denoted as CPFP.

The following sections use the previously introduced dataset from Section 5.1. As motivated in Section 5.4 the projects are split into test and training data according to the first release milestone, see Table 6.1 for the distribution data.

6.1. Within Project Prediction

If applying SFP within the same project as the metric data is collected, it is denoted as ”within project prediction”. All the reported results are acquired

Table 6.1.: Projects split overview

	Split		train		test		total
Project	test	train	fault free	buggy	fault free	buggy	bug rate
A	50	50	596	48	1.313	40	4,61%
K	40	60	941	252	1.579	123	14,89%
L	50	50	1.517	44	1.376	32	2,63%

using the dataset presented in Section 5.1, all Experiments were performed using the WEKA toolkit [104] and python scripts to automate the process. The presented content is based on Work by Altinger *et al.* [23].

The main motivation behind it is to establish a realistic setting when applying SFP to real world data. This requires considering the chronological order of the commits, thus when evaluating any predictor the test set is not allowed to randomly draw samples. The training set can be exposed to every resampling, filtering, *etc.* method known. Assuming historic data is available when starting SFP, as represented by the training set, and new commits are submitted over time, as represented by the chronological ordered test set. This ensures a correct simulation with the supplied dataset. Figure 6.1 shows the chosen experimental Setup. The dataset is split into test and training data, where the test data remains untouched.

From Section 5.4 the distribution of bugs is known. Therefore, splitting the dataset from Project A and L into 50% for training and 50% for testing is a valid approach, the same applies to project K at 40% training and 60% testing. A realistic utilisation allows starting SFP after an initial project development phase with already executed tests and bug fixes.

All reported performance values are described within Section 3.5.3, refering to equations 3.8, 3.9a, 3.9b,3.9c, 3.9d, 3.9e, 3.9f and 3.9h. Based on the definition by Weiss [126] the dataset suffers from a class distribution above 1:10, thus it is considered to be an imbalanced class problem. Following his

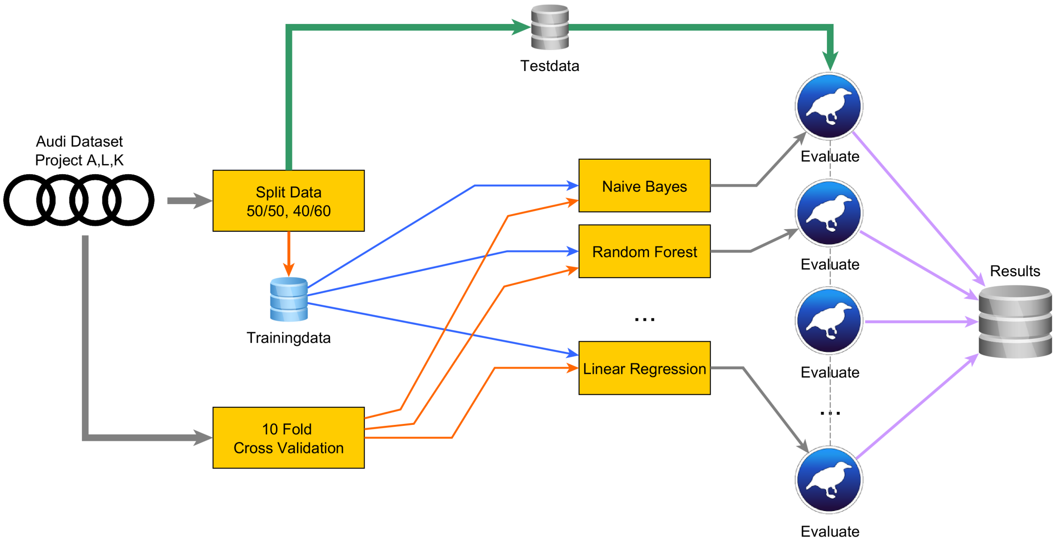


Figure 6.1.: Experimental Setup to evaluate SFP

arguments using F1 and G measure to compare the predictors performances is advised.

Assuming a simple predictor, set to always output "bug free", achieves a TPrate of 0,974, a precision of 0,939, an recall at 0,969 and AUC to be 0,5 when evaluated on project L. This is misleading, but possible due to the low number of bugs within the dataset as of its imbalanced class distribution nature. Thus it is important to limit measuring TP to the smaller class which should be taken into consideration when calculating the results.

Kim *et al.* [165] reports NB to be very robust against noisy data and multiple other publications (Menzies *et al.* [88], DÁmbros *et al.* [118]) list NB as the top performer. As presented in Table 3.3 NB along with RF and SVM achieve the highest AUC values upon the reviewed literature. Based on these findings Tables 6.2, 6.3 and 6.4 present the result of conducted experiments, using the dataset presented in Section 5.1 and these three well performing predictors among others. All experiments were carried out using the implementation supplied by WEKA [104] with no specific parameter tuning on a single

Table 6.2.: Predictive performance on project A using 50% to train and 50% to test.

	Project A, 50/50% Split							
	AUC	Accuracy	Recall	Precision	type I	type II	F1	G-Measure
Ada Boost M1 NB	0,987	0,95	1	0,25	0,05	0	0,4	0,974
Ada Boost M1 RF	0,995	0,952	1	0,258	0,048	0	0,41	0,975
SVM	0,776	0,983	0,563	0,5	0,009	0,007	0,529	0,717
LR	0,896	0,873	0,563	0,073	0,119	0,007	0,129	0,686
NB	0,952	0,875	1	0,119	0,125	0	0,212	0,932
RF	0,987	0,952	1	0,258	0,048	0	0,41	0,975
xgBoost	0,979	0,921	1	0,176	0,079	0	0,299	0,958

classifier.

Comparing these results, one can see that predictors in project A perform best in terms of AUC, accuracy and recall, whereas precision is rather low for project A and L. These results are slightly above other literature reports (see Table 3.2). This is mainly due to good measurement data as the prediction model is straight forward from the machine learning framework. Worth mentioning is that RF performs consistently on all three datasets, which is in line with other literature reports. In general the performance depends on the data, which is in line with Menzies [114] stating the choice of algorithm is strongly application dependant.

Table 6.5 reports on a 10 Fold Cross validation experiment with higher overall performance results than Tables 6.2-6.4. Some classifiers (RF, Ada Boost with NB) claim an F1 score close to 1.0, which is near the ideal and in line with the reported ”ceiling effect” as defined by Menzies *et al.* [135], [114]. Critical literature reports by Tan *et al.* [166] discovered Cross Validation may result in misleading higher precision when applied to an imbalanced

Table 6.3.: Predictive performance on project K using 40% to train and 60% to test.

	Project K, 40/60% Split							
	AUC	Accuracy	Recall	Precision	type I	type II	F1	G-Measure
Ada Boost M1 NB	0,862	0,92	0,653	0,816	0,024	0,056	0,726	0,781
Ada Boost M1 RF	0,936	0,899	0,588	0,738	0,034	0,067	0,655	0,729
SVM	0,64	0,874	0,294	0,809	0,011	0,115	0,431	0,453
LR	0,909	0,883	0,608	0,648	0,054	0,064	0,627	0,737
NB	0,917	0,875	0,747	0,59	0,084	0,041	0,659	0,816
RF	0,936	0,905	0,584	0,777	0,027	0,068	0,667	0,728
xgBoost	0,879	0,865	0,624	0,577	0,074	0,061	0,6	0,741

dataset. In addition, Cross validation does not represent the aimed realistic evaluation as defined earlier.

To compare findings the NASA KC2 dataset from the PROMISE repository [131] is treated the same way as the dataset from Section 5.1. Table 6.6 lists the achieved performance values. As there is no information of the chronological order or development milestones associated with the dataset the split share is set to 50% train and 50% testing, as these are common values reported in literature working with this NASA K2 dataset.

Recent work on a self tuning plugin called Auto-WEKA by Thornton *et al.* [167] it is very easy to optimize a machine learning algorithm which feeds the "ceiling effect" defined by Menzies *et al.* [135], [114]. During the majority of the experiments it performs quite well choosing meta learners for the presented automotive dataset. Thus this implementation makes it very easy to apply SFP to new projects even with little or no knowledge about machine learning algorithms.

Table 6.4.: Predictive performance on project L using 50% to train and 50% to test.

	Project L, 50/50% Split							
	AUC	Accuracy	Recall	Precision	type I	type II	F1	G-Measure
Ada Boost M1 NB	0,973	0,958	0,083	0,2	0,011	0,03	0,118	0,154
Ada Boost M1 RF	0,983	0,961	1	0,462	0,039	0	0,632	0,98
SVM	0,656	0,977	0,313	1	0	0,023	0,476	0,476
LR	0,989	0,961	1	0,457	0,039	0	0,627	0,979
NB	0,969	0,938	1	0,35	0,062	0	0,519	0,967
RF	0,984	0,961	1	0,462	0,039	0	0,632	0,98
xgBoost	0,980	0,958	1	0,444	0,042	0	0,615	0,978

6.2. Increasing Performance by Up-sampling Training Data

The previous Section 6.1 introduced SFP with no further modification of the training data. This section analysis of the predictive performance can be improved using over- and undersampling methods. The presented content is based on work by Altinger *et al.* [25].

The idea is inspired by literature as applying undersampling is suggested by Drummond *et al.* [75] and Khoshgoftaar *et al.* [116] to handle imbalanced class distribution. Further Japkowicz [69] studies an artificial two class dataset and presents over- and undersampling to overcome the imbalanced class problem.

6.2.1. Experimental Setup and Boundary Conditions

To handle modification of data the experimental Setup up is extended from Figure 6.1 to Figure 6.2 by adding a sampling filter between the training

Table 6.5.: Predictive performance on project K using 10 Fold Cross Validation

	Project K, 10 Fold Cross Validation							
	AUC	Accuracy	Recall	Precision	type I	type II	F1	G-Measure
Ada Boost M1 NB	0,996	0,986	0,942	0,965	0,005	0,009	0,954	0,967
Ada Boost M1 RF	1	0,993	0,975	0,979	0,003	0,004	0,977	0,986
SVM	0,714	0,909	0,437	0,899	0,007	0,083	0,588	0,607
LR	0,909	0,894	0,438	0,747	0,023	0,083	0,552	0,604
NB	0,947	0,842	0,908	0,484	0,145	0,014	0,631	0,867
RF	1	0,993	0,973	0,979	0,003	0,004	0,976	0,984
xgBoost	0,975	0,948	0,712	0,915	0,01	0,043	0,801	0,828

data and the evaluation. As known from Section 5.3 the metric data is non Gaussian distributed, thus the WEKA packages ”unsupervised instance resampling” filter need to be used, as the ”supervised” instance relies on Gaussian distribution. Table 6.8 the parameters for the under- and over-sampling. Further the SMOTE filter by Chawla *et al.* [168] is applied as it is designed specifically for imbalanced class problems.

Table 6.7 lists the classifiers used to predict bugs. LR is added for comparability reasons as multiple literature (*example given* Ostrand *et al.* [78], Weyuker *et al.* [91], Zimmermann *et al.* [93]) use this learner, though the data is non Gaussian and using LR is not advised. Weiss [126] recommends disclaiming the ’conquer and divide’ based classifiers, such as SVM or trees, as they are affected by the imbalanced class distribution. Their data is fragmented and contains rare samples which might lead to over interpreting them. Despite their good performance reports in other studies (Singh *et al.* [109], Hsieh *et al.* [169]) rbfSVM[1] and dcSVM are evaluated. RF is reported to be very robust (Guo *et al.* [76]) along with NB (Catal *et al.* [102]). Early work

[1] SVM with radial basis function kernel

Table 6.6.: Predictive performance on project NASA KC2 using 50% to train and 50% to test.

	Project NASA KC2							
	AUC	Accuracy	Recall	Precission	type I	type II	F1	G-Measure
Ada Boost M1 NB	0,591	0,678	0,208	0,84	0,015	0,307	0,333	0,343
Ada Boost M1 RF	0,633	0,651	0,129	0,813	0,011	0,337	0,222	0,228
SVM	0,533	0,636	0,079	0,8	0,008	0,356	0,144	0,147
LR	0,536	0,667	0,198	0,769	0,023	0,31	0,315	0,328
NB	0,723	0,678	0,208	0,84	0,015	0,307	0,333	0,343
RF	0,674	0,651	0,129	0,813	0,011	0,337	0,222	0,228
xgBoost	0,591	0,678	0,218	0,815	0,019	0,303	0,344	0,356

by Drummond *et al.* [75] reports on successfully using C4.5 for imbalanced class problems. WEKA offers an implementation of C4.5 called J48.

The majority of classifiers implementation is used from the WEKA toolkit, the others are implemented in R and being bridged into WEKA using the *rJava* plugin. This ensures a constant workflow as the automation is implemented using WEKA's Java API. To keep the results transferable no specific tuning is performed for a single classifier, thus all settings are default as defined by WEKA.

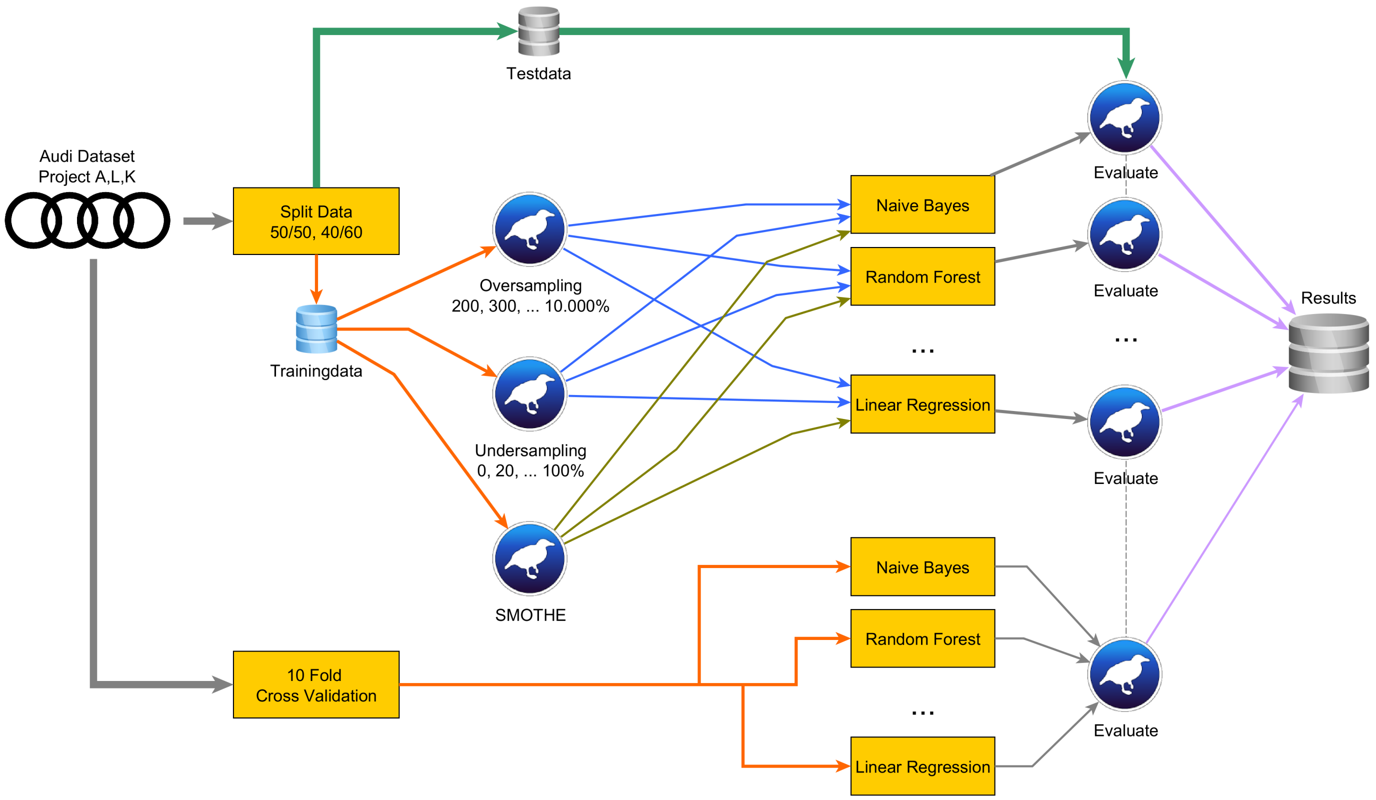


Figure 6.2.: Experimental Setup to evaluating over- and undersampling effects on SFP

Table 6.7.: Overview to used classifiers and their origin.

Learner	Implementation	Literature
Ada Boost NB	WEKA	[104]
Ada Boost RF	WEKA	[104]
dcSVM	R, Package *SwarmSVN*	[169]
J48	WEKA	[104]
Jrip	WEKA	[104]
LR	WEKA	[104]
NB	WEKA	[104]
random tree	WEKA	[104]
rbfSVM	WEKA	[104]
RF	WEKA	[104]
rPART	R, Package *rPart*	[170]
xgBoost	R, Package *xgBoost*	[171]

The 'unsupervised' resampling filter adds or removes samples by randomly copying and respectively deleting, entries from the input data. It does expect a share as parameters, see Table 6.8. Using 50% means to cut in half, 300% means to triple the number of samples from the original dataset. 100% represents the same amount as in the input data, but the distribution may be affected by the resampling algorithm.
The experiments implementation generates the data samples first and evaluates the exact same data on all classifiers, thus there is no influence upon random drawn sample sets, see Algorithm 1.

6.2.2. Results

As suggested by Weiss [126] the F1 score, see Equation 3.9f, is used to report the classifiers performance. F1 is based on recall and precision which can be targeted to the minority class and is not deformed by the number of TP, rooted in the imbalanced class distribution, such as AUC or TPrate. The same is valid for the G-measure 3.9h as it favours the FP measurement.

Algorithm 1 The experiments pseudo-code

```
$input\_files$                                              ▷ the projects data
$Classifiers$                                               ▷ the classifiers to evaluate
$resample\_filter$                                          ▷ none, Resample, SMOTE
$resample\_settings$                                        ▷ none, Resample share
foreach $file \in input\_files$ do
    $orig\_data \leftarrow open(file)$
    $train\_data \leftarrow split(orig\_data, split\_percentage)$
    $test\_data \leftarrow split(orig\_data, 1 - split\_percentage)$
    foreach $filter \in resample\_filters$ do
        foreach $settings \in resample\_settings$ do
            $sample\_train\_data \leftarrow sampleData(filter, settings, train\_data)$
            foreach $classifier \in Classifiers$ do
                $train(classifier, sample\_train\_data)$
                $evaluate(classifier, test\_data)$
            end foreach
        end foreach
    end foreach
end foreach
```

Table 6.8.: Parameters and their ranges used during experiments.

Parameter	min	step	max
undersampling %	0	10	100
oversampling %	200	100	900
oversampling %	1.000	500	3.500
oversampling %	4.000	1.000	10.000

G-measure is not reported as its resulting graphs are very similar to F1.

Table 6.9 summarises our findings compiled from figures 6.3 to 6.5. Using u means it is undesirable as the performance is noisy over the sampling sweep which is the case for most of the boosting algorithms. A positive influence, marked with p, represents a consistent result, higher than when using the original data. A negative influence, marked with n, is set if the performance is lower than on the original data. If there is neither a better nor a worse performance through the sampling sweep o is set. In total, strong undersampling (20 - 30%) is negative across all classifiers and projects. In comparison, there is no overall single classifier to benefit all datasets and sampling settings. Table 6.10 sums up the decision of Table 6.9 and presents an even distribution between u - undesirable and o - no effect, but also a similar number of p - positive effects.

Figure 6.3 reports the F1 performance of all nine predictors on project A. Clear dcSVM achieves the highest performance, but is heavily influenced by under- and oversampling. rPART and xgBoost benefit from high oversampling, LR delivers the lowest performance only increasing marginally above the original data setting. Overall SMOTE archives equal to slightly better results across all classifiers than the untouched data, but never the highest values. The majority of classifiers (Ada boost, random trees, *etc.*) becomes unpredictable as their results strongly depend on a single setting.

In contrast to project A results presented in Figure 6.3 the results for project K in Figure 6.4 dcSVM and rbfSVM are the worst performing predictors. NB along with rPart increase performance with inclining oversampling. The other classifiers suffer from a lot of noise, hence their predicting performance

Table 6.9.: Results Overview to Classifiers performance on over- and undersampling

	Undersampling			Oversampling			Oversampling		
	40 - 90%			200 - 1000%			> 1000%		
classifier	A	L	K	A	L	K	A	L	K
Ada Boost NB	u	u	u	u	n	u	u	p	u
Ada Boost RF	u	u	o	u	p	o	u	p	o
dcSVM	p	n	n	p	n	p	p	n	n
J48	u	n	o	p	u	o	p	u	o
Jrip	u	o	n	p	o	u	p	o	u
LR	n	p	o	p	o	o	o	o	o
NB	n	n	n	o	p	p	p	p	p
random tree	u	u	u	u	u	u	u	u	u
rbfSVM	p	n	p	p	p	n	p	p	n
RF	n	o	o	p	o	o	p	o	o
rPART	u	n	n	o	o	p	p	o	p
xgBoost	u	o	n	u	o	o	u	o	o

strongly depending on the Resample settings.
Similar to Project A in Figure 6.3 the best performing classifiers on project L in Figure 6.5 are the SVM based algorithms. In contrast to Project A and K LR achieves a better performance than most of the other classifiers. Despite random trees, Ada boosting and J48 the majority of classifiers delivers stable performance figures.
Selecting a single classifier NB Figure 6.6 displays the dedicated performance measurements defined in Equation 3.9. Clearly one can see recall and precision can not be improved at the same time. One has to choose which of them to increase by over- or undersampling the training data. Performance ratios as F1 and G-measure building on top of recall and precision show this tendency in comparison to AUC which remains constant.

Summarizing all results there is no clear overall trend on performance improvements for all classifiers and projects. It is possible to specifically

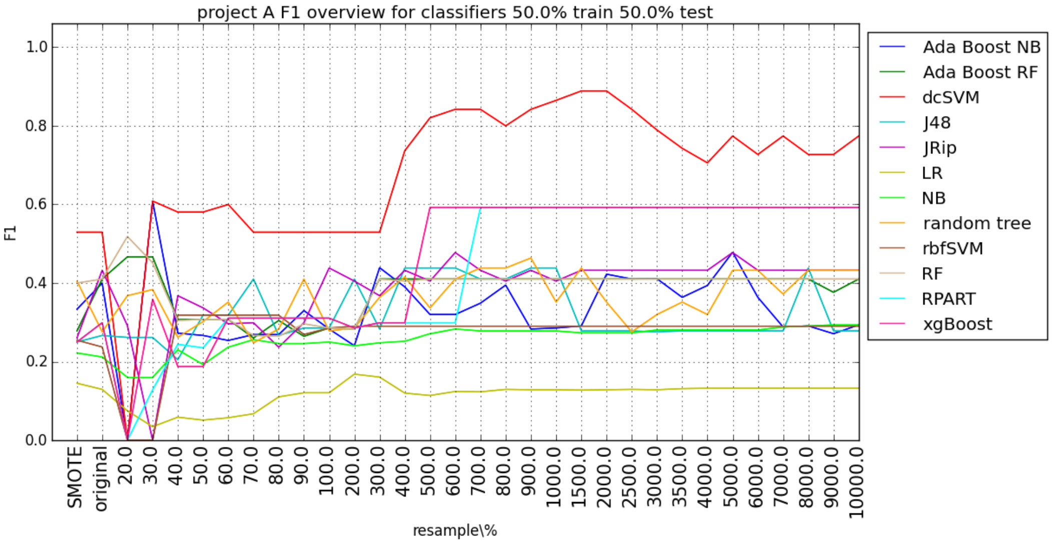


Figure 6.3.: Performance overview for Project A, using 50% to train and 50% to test

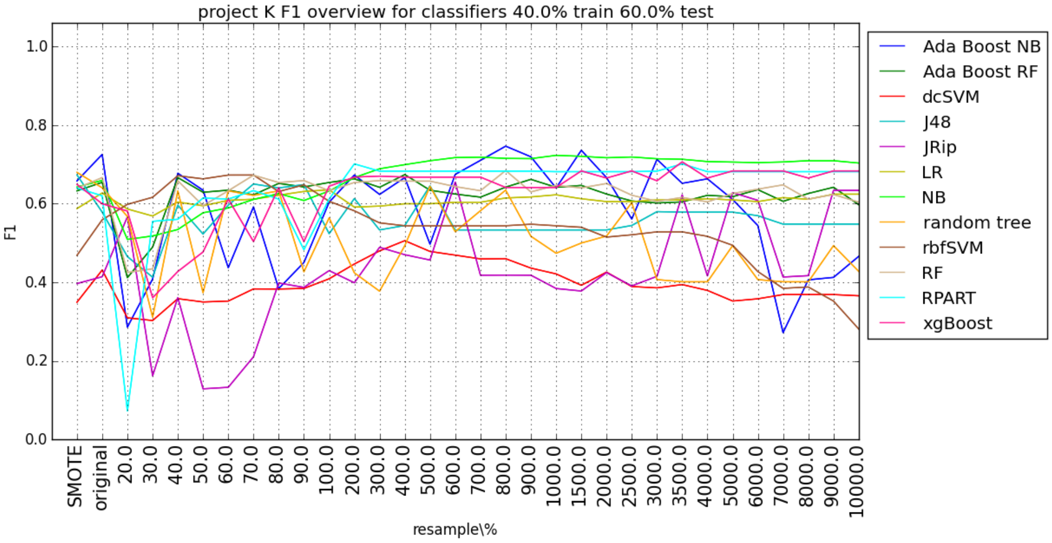


Figure 6.4.: Performance overview for Project K, using 40% to train and 60% to test

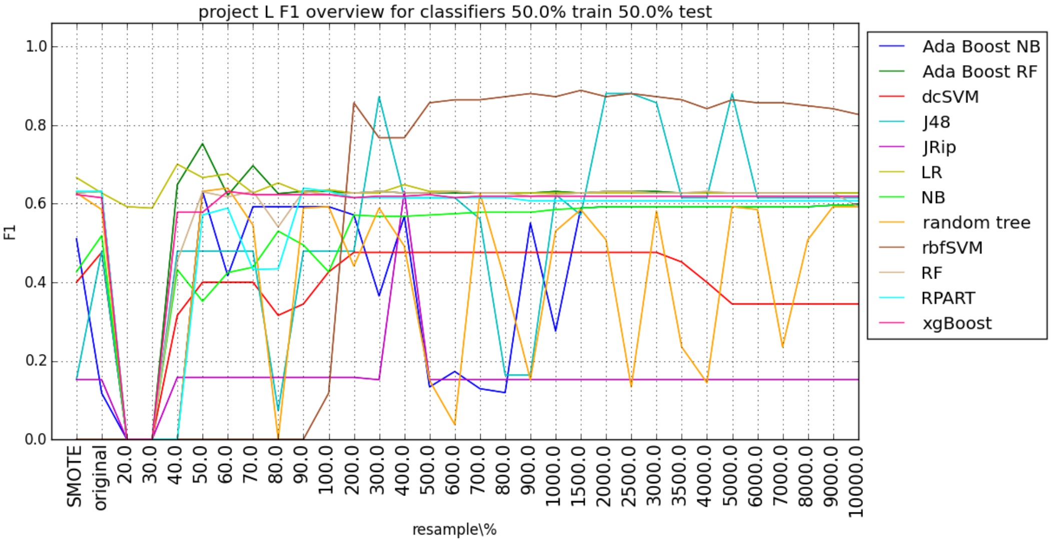


Figure 6.5.: Performance overview for Project L, using 50% to train and 50% to test

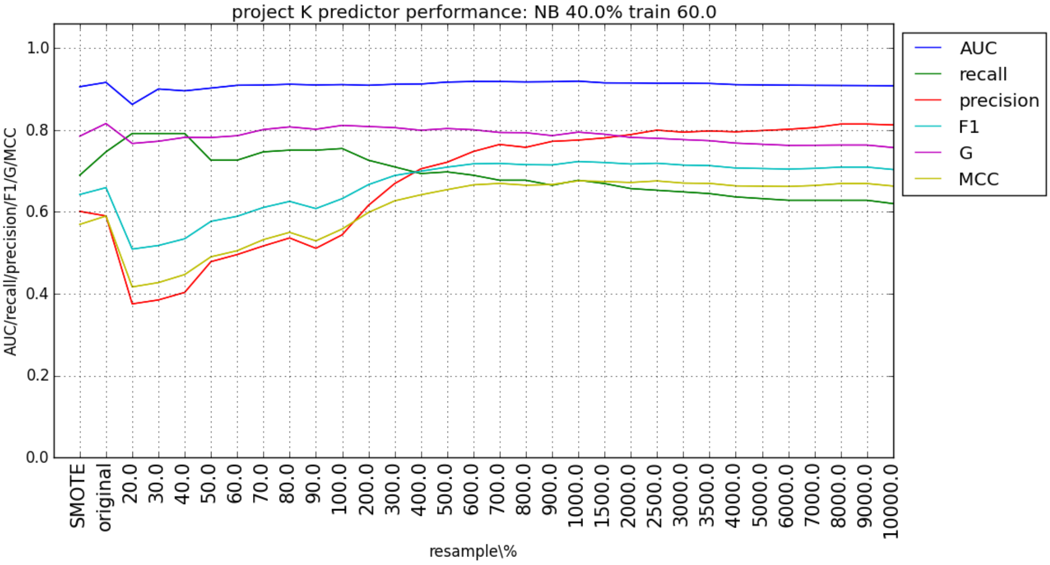


Figure 6.6.: Performance NB on project K, using 40% to train and 60% to test

Table 6.10.: Summary to Table 6.9

	Undersampling			Oversampling			Oversampling			
	40 - 90%			200 - 1000%			> 1000%			
type	A	L	K	A	L	K	A	L	K	Sum
u	7	3	2	4	2	3	4	2	3	30
p	2	1	1	6	3	3	7	4	2	29
n	3	5	5	0	2	1	0	1	2	19
o	0	3	4	2	5	5	1	5	5	30

improve one classifier for a given dataset by applying (higher) oversampling. Recall and precision can not be tuned at the same time by only applying resampling.

6.3. Cross Project Fault Prediction

At the early stages of every project there may not be enough historic data to set up a SFP, thus reusing the data from a similar project to train the predictor leading to the definition of 'Cross Project Prediction', CPFP. The presented content is based on Work by Altinger *et al.* [24].

The experiments Setup is adopted from Figure 6.2 to Figure 6.7, the evaluation process is extended from Algorithm 1, by adding the selection of a different project as training data, which leads to Algorithm 2. Pre filtering is performed by SMOTE and the 'unsupervised Resample' filter with a share sweep identical to Section 6.2.1, see Table 6.8. As literature (Herbold [122], Zimmerman *et al.* [111], Turhan *et al.* [110]) on CPFP uses LR, NB, SVM, *etc.* the same classifiers as in the previous section are used, see Table 6.7. In total 2.304 experiments are performed and evaluated.
All conducted CPFP settings perform rather poorly, although as suggested by Zimmermann *et al.* [111] using most similar project settings (tools, methods, reporting, ...) are fulfilled for the used dataset. If the criteria is lowered to require recall and precision above 0.65, there are 27 successful of whom rPART delivers half of them. Still this is a very low success rate of 1.17%

Algorithm 2 The CPFP experiments pseudo-code

```
$input\_files$                                        ▷ the projects data
$Classifiers$                                         ▷ the classifiers to evaluate
$resample\_filter$                                    ▷ none, Resample, SMOTE
$resample\_settings$                                  ▷ none, Resample share
foreach $file \in input\_files$ do
    foreach $comp\_file \in input\_files$ do
        if $file \neq comp\_file$ then
            $train\_data \leftarrow open(file)$
            $test\_data \leftarrow split(comp\_file)$
            foreach $filter \in resample\_filters$ do
                foreach $settings \in resample\_settings$ do
                    $sample\_train\_data \leftarrow sampleData(filter, settings, train\_data)$
                    foreach $classifier \in Classifiers$ do
                        $train(classifier, sample\_train\_data)$
                        $evaluate(classifier, test\_data)$
                    end foreach
                end foreach
            end foreach
        end if
    end foreach
end foreach
```

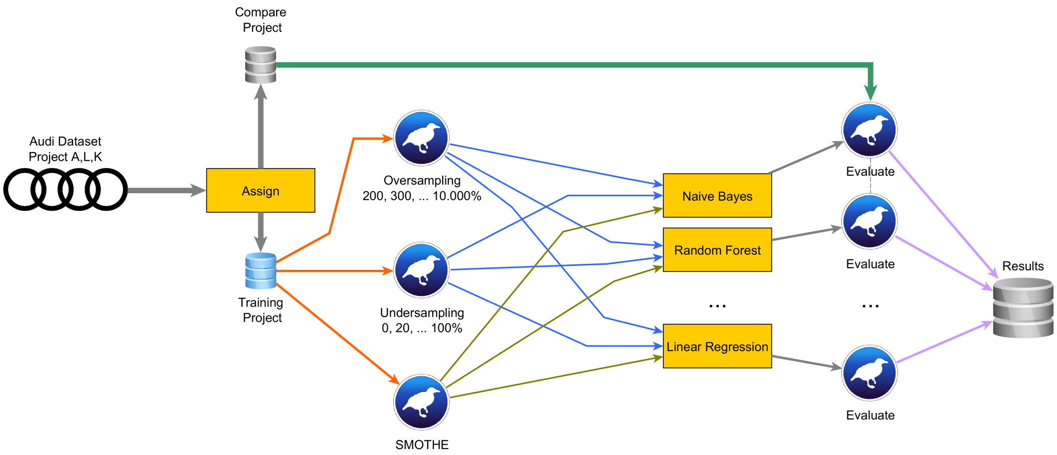


Figure 6.7.: Experimental Setup to evaluating over- and undersampling effects on CPFP

where K is always used to train and L to test. Figure 6.8 presents the F1 score for all evaluated classifiers on this setting. Clearly there is no overall best performing classifier. rPart and J48 remain consistent with higher sampling, boosting algorithm showing a negative tendency, all other classifiers deliver unpredictable performance across the resampling sweep.
Similar to Section 6.2.2 Figure 6.9 visualises recall and precision can not be increased at the same time, one either needs to decide which of these performance measures to favour and improve resampling. The overall performance increases with higher sampling, but might not be consistent. This is in line with findings by Turhan *et al.* [110] where they discovered no increasing predictive performance with larger trainings sets.

Following the definition used by Zimmerman *et al.* [111] where recall and precision need to be above 0,75 to count the experiment as positive, none of the conduced experiments are successful.

To gain insight, a Kendal τ correlation analysis similar to Section 5.3 between project A and K is performed, see Table 6.11. As project K and L perform better in CPFP, a Kendal τ correlation analysis is performed between these two, see Table 6.12. Clearly there is a higher number of correlating factors between project K and L than between K and A. This is a similar result to

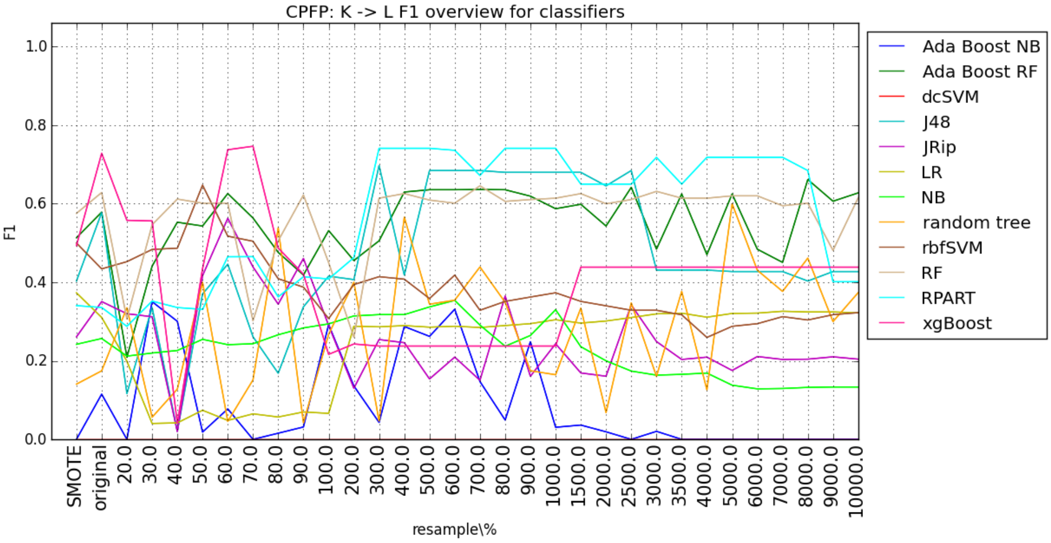


Figure 6.8.: CPFP using Project K to train and project L to test, all classifiers F1 performance.

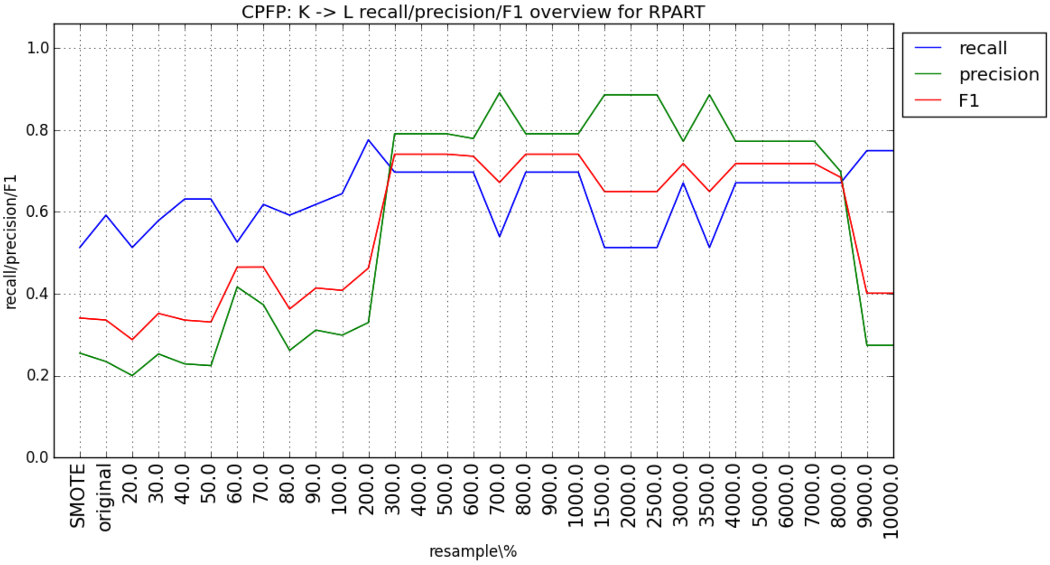
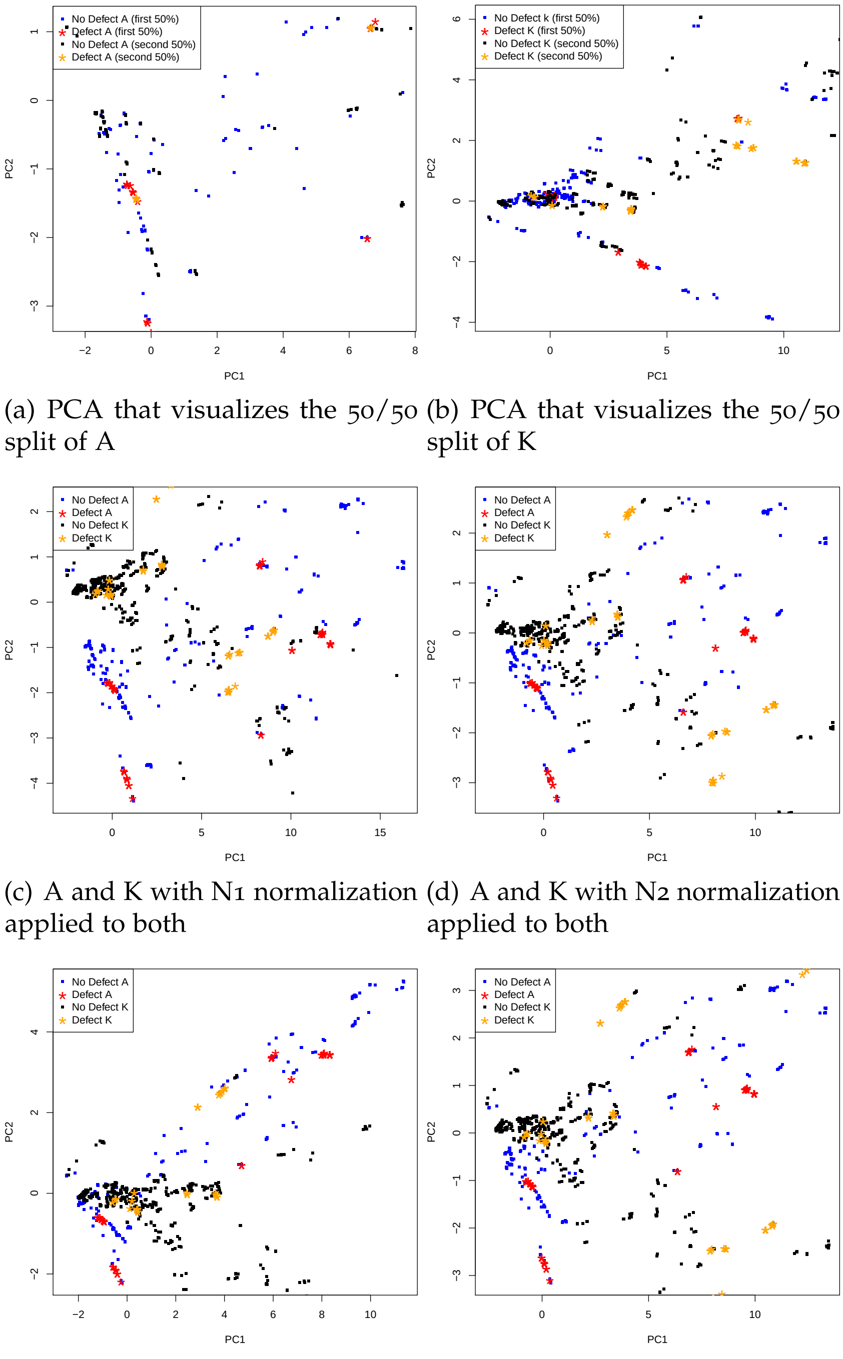


Figure 6.9.: CPFP using Project K to train and project L to test, rPART as classifier.

Section 5.3 where significant correlation between software metrics and the bugs indicate whether SFP can be successful or not.
Altinger *et al.* [24] performed a PCA to gain insight into the bug distribution across the software metrics between the projects, see Figure 6.10. Clearly the bugs occur in separate metric regions, thus resulting in similar findings than the correlation analysis in Tables 6.11 and 6.12.
As suggested by Herbold [122], Altinger *et al.* [24] perform nearest neighbour filtering between the datasets to pre filter the training data. Even applying all recommended methods, it is not possible to extend the CPFP results beyond 0,36 F1.

CPFP achieves a very low performance, thus it is not applicable, although the project settings (development guidelines, tools, company settings, *etc.*) are the most similar they can be. This may be caused by non-correlating software metrics between the projects and the non-overlapping regions where bugs occurred.



(a) PCA that visualizes the 50/50 split of A

(b) PCA that visualizes the 50/50 split of K

(c) A and K with N1 normalization applied to both

(d) A and K with N2 normalization applied to both

(e) A and K with N4 normalization applied to A

(f) A and K with N5 normalization applied to A

Figure 6.10.: The PCA is performed with all data from project A and K. The first two principle components explain between 89%-92% of the variance. Figure taken from Altinger *et al.* [24]

Table 6.11.: Kandals τ correlation analysis for metrics between error classes of project A and K, strong correlations are marked in bold.

2

		Project K										
		author	LOC	CC	Hv	Hd	He	LOC_add	LOC_remove	nfunctions	commit_age	num_commits
Project A	author	nan	-0,01	0,04	0,05	0,03	0	0,1	0,13	0,07	-0,1	-0,08
	LOC		0,55	0,34	0,29	0,59	0,46	0,07	0,07	0,46	0,14	0,18
	CC			0,23	0,17	0,47	0,35	0,08	0,05	0,35	0,18	0,09
	Hv				0,25	0,51	0,42	0,03	0,02	0,37	0,21	0,18
	Hd					0,14	0,04	0,13	0,02	0,17	-0,07	-0,17
	He						0,31	0,06	0	0,4	0,15	0,08
	LOC_add							-0,03	0,07	-0,01	-0,04	-0,01
	LOC_remove								0,02	-0,03	-0,04	-0,02
	nfunctions									nan	nan	nan
	commit_age										0,33	0,24
	num_commits											0,24

Table 6.12.: Kandals τ correlation analysis for metrics between error classes of project L and K, strong correlations are marked in bold.

		Project K										
		author	LOC	CC	Hv	Hd	He	LOC_add	LOC_remove	nfunctions	commit_age	num_commits
Project L	author	0,11	-0,07	-0,12	-0,11	0,15	0	-0,02	0,05	-0,14	-0,15	-0,15
	LOC		**0,75**	**0,68**	**0,7**	0,19	0,65	0,09	0,16	0,68	**0,7**	**0,7**
	CC			**0,76**	**0,77**	0,16	**0,7**	0	0,04	**0,75**	**0,76**	**0,76**
	Hv				**0,76**	0,16	0,64	0,04	0,09	**0,76**	**0,79**	**0,79**
	Hd					-0,35	0,1	-0,2	-0,23	0,12	0,17	0,17
	He						0,43	-0,08	-0,05	0,55	0,58	0,58
	LOC_add							0,1	0,13	0,05	0,06	0,06
	LOC_remove								0,15	0,1	0,11	0,11
	nfunctions									nan	nan	nan
	commit_age										**0,91**	**0,91**
	num_commits											**0,92**

7. Conclusion

This chapter summarizes the presented work in Section 7.1, it presents Threats to Validity in Section7.2 and finishes with further research questions in Section 7.3.

7.1. Summary

This work presents analysis of software development within the automotive industry. A conducted survey presents Matlab Simulink to be the most common tool in designing models which will be used to automatically derive code (RQ1). The existing restrictive coding standards successfully prevent typical bug categories. Gathering software metric data from model derived source code is possible. Further extracting bug information from an ITS system and tracing bugs across multiple revision benefits from restrictive development guidelines and small code changes between revision. A public available dataset was compiled from three real world software projects being the first to offer software metrics enabling SFP from the automotive industry. The conduced correlation analysis between the metrics and the bugs did not show any connection (RQ3, RQ 3.2) between those. Software volume metrics do correlate with each other, but author information or code chrun metrics are independent (RQ 3.1). The applied SFP study benefits from high quality measurement data (RQ 2.1) as the bug information from the ITS is very reliable. The achieved predictive performance is higher than reported in literature although using standard settings for all used classification algorithms (RQ2). There is no clear best performing classifier across all conducted experiments, although RF and NB seem to be more stable and achieve true positive rates beyond 0.95 (RQ4). These results still need to consider the strong imbalanced datasets, where even a simple predictor will achieve similar values. Further the predictive performance can be increased using higher oversampling (RQ 2.3), but the results strongly depend on the project and the used classifiers as not all of them perform stable when applying an under- and oversampling sweep. A conduced CPFP experiment does not perform well (RQ 2.4) as there is only one setting where a project successfully predicts bugs within another one as the metric correlation is weak and the bug distribution scatters too far.

7.2. Threats to Validity

Following Perry *et al.* [172] one should give three threats to validity: Construct validity, internal validity and external validity.
The presented questionnaire survey (Section 4.1) consists of a limited number of questions, thus only covering a narrow scope of tools and methods. The findings are related to the automotive industry and should not be generalized to other software industries. The number of respondents at 68 is rather low, thus the survey is not representative. Furthermore the respondents were asked to distribute the questionnaire among their colleague, hence no initial company distribution is known. The answers were collected anonymously, therefore double responses can not be neglected. As the author's contact details are related to one OEM specific "tuned" answers to company dependent processes are possible.

The presented (Section 5.1) and further used dataset only contains data from three projects. Typical to automotive projects they cover very specific topics, therefore influences and limitations can not be excluded, thus the scope is limited. As the software is automatically generated using a template based code generator a major release upgrade may just influence the code metrics as the generated code significantly changes. Thus the generalizability of the findings is limited.
As the dataset contains a proprietary code the repeatability of the Error class analysis (Section 5.5) is limited. The discovered findings may be influenced by the nature of the studied projects as they mainly consist of boolean logic statements.
According to the SFP and CPFP studies (Section 6) no single classifier tuning is performed. Thus the used classifiers may not use their full potential and other researchers can achieve better overall performances. As CPFP is not a core focus of this thesis only tests with state of the art methods were

performed, thus the ”non-applicable assumption” could be disproved by another method.

The best performance on the resampling experiments (Section 6.2) is achieved using high oversampling. This is computationally possible as the case studies dataset is small, thus the findings may not be transferable to others due to computational limits.

7.3. Further Research

Zheng *et al.* [129] present a database hosting trained classifiers for a very wide range of software projects. The authors use a similarity metric to select the best fitting training set. This seems to be a very promising approach to CPFP, which requires adding more automotive datasets to publicly available databases. The connected research question: **Are imbalanced datasets similar enough to share trained predictors**?

The presented dataset from Section 5.1 suffered from an inverse chronological order, still Altinger *et al.* [23] was able to achieve successful SFP predictions. Altinger *et al.* [25] use a fixed dataset and achieves a remarkable higher prediction. This leads to another research question: **Is time dependant between the measurement updates on matter in SFP**?

Le *et al.* [173] presents work on automated software repair which suffers from high computation requirements with low success rates as all possible mutations of statements need to be processed. SFP may help in identifying possible errors and limit the scope in searching for repair mutations. This leads to a research question: **Can SFP predict bugs at a sufficient precision to successfully guide software repair to pass TCs**?

Many agile software projects use nightly builds compiled of multiple new code contributions. If the build fails there is no information upon the hook causing the failure. A simple approach would be to incrementally revert the changes and repeat the build and rerun the test suite. This is a computationally exhaustive method. This leads to the last new research question: **Can SFP predict which commit is the most likely to contain the fault?** Thus the revert can be set to its prior revision which will reduce the computational efforts.

Appendix

Appendix A.

Publication List

As requested by the curriculum and the statutes of the doctoral school this chapter will list all publications and their relation to this thesis.

A.1. Authors Publications Relevant to this Thesis

This section presents this thesis author's work related to the thesis content, an overview can be extracted within Table A.1. The following four publications were presented at international workshops and conferences, all of them are peer reviewed.

Altinger *et al.* [13]: main work by me, Wotawa supported in defining the formulation of the questions, other authors contributed single phrases and performed reviews.
Altinger *et al.* [23], main work by me, Dajsuren supplied the Matlab Simulink Model measurement tool and help upon model metric data interpretation, other authors contributed single phrases and performed reviews.
Altinger *et al.* [24], main work by me, Herbold conducted the PCA and executed the CPFP along with discussing the research questions and results, other authors contributed single phrases and performed reviews.
Altinger *et al.* [26], main work by me, other authors did contribute single phrases and performed reviews. Altinger *et al.* [22][1]: main work by me and Bock, where Bock contributed the second survey and the associated analysis. Wotawa and German contributed single phrases and performed reviews.

[1] This publication is submitted to review and is not published at date of release of this thesis

Altinger *et al.* [25]: main work by me, Herbold supported with defining the research questions and result interpretations, other authors contributed single phrases and performed reviews.
During all publications Professor Wotawa worked as a interlocutor to discuss the research questions and performed reviews and minor textual formulations.

A.2. Authors Publications not Relevant to this Thesis

During my work on RoboCup Middle Size Liege:
Altinger *et al.* [174] together with my fellow student colleague and ”Mostly Harmless” team mate Stephan Mühlbacher-Karrer. He worked on the electronic circuit design, I worked on the measurement software, both of us performed the experiments, other authors contributed single phrases and performed reviews.
Kollar *et al.* [175] during which my ”Mostly Harmless” team mate Michael Kollar set up the system and performed the experiments, I assisted on the research questions and the publications structure.

During my work on self driving cars at AUDI AG:
Altinger [176] as my master thesis where I developed the system, set up the car and conducted the experiments. Wotawa and Steinbauer served as supervisors and interlocutor to define the research questions.
Altinger *et al.* [177] where I act as the author and presenter of the publication along with contributing to the implementation along with a team of ten engineers.
Ibisch *et al.* [178], Ibisch *et al.* [179], Ibisch *et al.* [180] is work by my colleague André Ibisch where I supported the implementation of the presented system and helped in defining the research questions and conducting the experiments along with the result interpretations.
Sippl *et al.* [181] is work by my colleague Christoph Sippl where I served as a spinning partner to derive the presented system and define the research questions. In addition I wrote parts of the related work and conducted reviews.

During my work on Simulink model analysis:
Olszewska *et al.* [182] where I conducted reviews and supplied measurement data along with their interpretation.

During my work on organizing Workshops:
Together with my colleagues Yanja Dajsuren and Miroslaw Staron I organized the Workshop on Automotive Software Architectures (WASA) 2015 and 2016 during the WICSA/CompArch conference. Kruchten *et al.* [183] hosts the proceedings where I performed reviews.
Together with my colleague Bernhard Peischl I did organize the workshop on Digita Eco Systems held at the ICTSS conference in 2016.

Patents during my work at AUDI AG:
Togehter with my working colleague Florian Schuller I released two patents, Altinger and Schuller [184] and Schuller and Altinger [185] covering parts of our work on self driving cars. The core Ideas were developed during discussions and implementations as joint work between the two of us. The patents Editor is LINDNER and BLAUMEIER and patent agent from Munich.

Invited Talks:
Altinger [186] is an invited talk covering the presented work within this thesis and my work on self driving cars at AUDI AG.

Table A.1.: Annotated list of pulications. Chapters refer to this thesis.

Citation	contributor		Others	Chapters
	main	support		
[13]	Altinger	Wotawa	review, comments	4.1
[23]	Altinger	Dajsuren	review, comments	4.2, 5.1, 5.2, 5.3
[24]	Altinger	Herbold	review, comments	6.1, 6.3
[26]	Altinger		review, comments	5.5
[25]	Altinger	Herbold	review, comments	6.2

Appendix B.

Questions from the Survey

The questions used during the questionnaire survey are listed below. They are available as pdf, see http://www.ist.tugraz.at/_attach/Publish/AltingerHarald/Survey_automated_testing_enu.pdf, and online via Surveymonkey®, see https://de.surveymonkey.com/s/survey_testautomation.

Survey on automated Testing @ automotive domain

The results from this question form will be used to set up a survey on the use of automated testing methods within the automotive domain. The survey will be handed in for publication on IEEE intelligent vehicle symposium 2015.

For all questions it is possible to check multiple options if applicable or / and use the other field to add comments. Some questions have embedded help text to clarify or explain their answer choices.
All collected data will be treated anonym, it is not required to name any company name, department, etc.

This form is fillable, please send it back to the author via email.
If you have any questions, please do not hesitate and contact the author via e-mail: harald.altinger@student.tugraz.at or harald.altinger@audi.de

TU Graz
Graz University of Technology

iST

Audi
Electronics Venture GmbH

Company definition	OEM
	Supplier (Tier 1-3), engineering service provider
	University / Research facility
	Other
What is the area of work within your department?	Research
	Pre development
	Series development
	Other
What is your job description?	
What is the purpose of your department?	Pure software development
	Pure hardware development
	Soft- & and hardware development
	Testing
	Other
Which development target(s) is your department using?	Whole car (integration)
	ECU (or embedded controller)
	Software library
	PC / smartphone / commercial devices
	Rapid prototyping devices, e.g. AutoBox
	Other
Which time response does your target application require?	Hard real time
	Soft (or firm) real time
	None
	Other

if the department is software related:

Which type of software is your department developing?	Algorithmic Control program Function development Other
Which languages are in use to create models or develope software within your department?	C/C++ Java / C# Spark / ADA / B / Z Scade Modelica Matlab / Simulink LabView UML / SysML Other
Does your department use development frameworks?	Commercial Open source Self developed (within department or company) Name of framework

---

The following questions are testing related:

Which kind of code access does your department have while testing?	White box Black box Grey box Other
Which type of testing method is in use within your department?	Fuzzy testing Model driven testing / model based testing Mutation testing Random testing Test driven development Unit testing Other
Which test automation methods are in use within your department?	Automated test case generation Automated testing / automated testcase execution Test-suite management Manual testing Other
Name of test automation tool	

The following questions are related to tools in use:

Which type of license do the implemented testing tools have?	Open Source Commercial Self developed
What is your department's application area for those tools?	Static code analysis Code review Code coverage Formal methods Test automation Other

---

The following questions are related to specification (requirements)

How does your department compose requirements? please do answeare this even if your department only receives requirements.	Textual - natural language Textual - natural language (reduced vocabulary, e.g. templates) Textual - formal language (mathematical, logics, ...) Textual - pseudo code Graphical - functional diagrams Graphical - state diagrams Graphical - sketches Other
Which kind of tools is in use when composing requirements?	Requirement management system Word processing (any editor) Diagram editor Other
name of tool	

The following questions are related to technical standards

Which standards have to be considered when developing software within your department?	ISO 26262 ISO 9001 SPICE AUTOSAR Other

The following questions are related to the work distributed among the personnel.

Which person performs the testing?	Performed by developer Performed by specifier Performed by 3rd party Other

---

The following questions are related to testing stages

When performing testing, what is the primary scope?	Unit (module) testing Integration testing (whole) System testing User acceptance testing Other
During which development stage(s) is testing performed within your department?	Simulation MiL ... Model in the Loop SiL ... Software in the Loop PiL ... Processor in the Loop HiL ... Hardware in the Loop Car final test Other
What is the basic testing characteristic in use?	Positive testing False testing System behavior outside specification testing

The following questions are related to experience gained when using testing (automated or manual) and based on your outlook about testing.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
The costs (time & money) of testing are worth it (improve code-quality, reduce bugs, etc.)					
The ROI on testing is possible (less maintenance due to bugs, reduced development time, etc.)					
Evaluating and setting up automated testing tool(chains) takes too much time.					
We plan on (increasing our) investing in testing automation within the next year.					
Creating Models, defining constraints and testing parameters requires more time than the actual implementation.					

Is there anything you would like to add or clarify for your answers?

Appendix C.

Acronyms

ACC	Adaptive Cruise Control
ACO	Ant Colony Optimization
ADAS	Advanced Driver Assistance Systems
AEV	Audi Electronics Venture GmbH
API	Application Programming Interface
ASIL	Automotive Safety Integrity Level
AUC	Area Under Curve
AUTOSAR	AUTomotive Open System ARchitecture
CAN	Controller Area Network
CC	Cyclomatic Complexity by McCabe
CVS	Concurrent Versions System
CPFP	Cross-Project Fault Prediction
CPU	Central Processing Unit
DS	Discrimant Statistic
DT	Decision Trees
ECU	Electronic Control Unit

EnProVe	EntwicklungsProzess Verbesserung, german: Development Process Improvement
ESD	Electrostatic Discharge
FN	False Negative
FP	False Positive
GM	General Motors
GLM	Generalized Linear Models
GPS	Global Positioning System
GUI	Graphical User Interface
HE	Halstead Effort
HV	Halstead Volume
HiL	Hardware in the Loop
IDE	Integrated Development Environment
ITS	Issue Tracking System
KBA	Kraftfahr Bundesamt (English: German Federal Motor Transport Authority)
CK	Chidamber Kemerer [60] OO metric
Kessy	Keyless Entry Start and Exit System
LOC	Lines Of Code
LR	Logistic Regression
MiL	Model in the Loop
MAAB	Mathworks Automotive Advisory Board
MARS	Multivariate Adaptive Regression Splines

MISRA	Motor Industry Software Reliability Association
ML	Machine Learning
MOST	Media Oriented Systems Transport
MsOffice	Microsoft Office Suite
MSR	working conference on Mining Software Repositories
NADA	National Automobile Dealers Association
NASAMDP	NASA metric data program [133]
NB	Näive Bayes
NBR	Negative Binomial Regression
NHTSA	National Highway Traffic Safety Administration
NN	Neural Network
OEM	Original Equipment Manufacturer
OO	Object Orientated
OS	Operationg System
PCA	Principle Component Analysis
PD	Pre Development
PiL	Processor in the Loop
PLC	Product Life Cycle
RD	Research and Development
regex	regular expressions
RF	Random Forest
ROI	Return Of Investment

RTE	Run Time Environment
RU	Research
SCM	Source Control Management
SD	Series Development
SES	Supplier (Tier 1-3), engineering service provider, software vendor
SFP	Software Fault Prediction
SiL	Software in the Loop
SIM	Subscriber Identification Module
SIR	Software-artifact Infrastructure Repository
SOP	Start Of Production
SQL	Structured Query Language
SVM	Support Vector Machine
SVN	SubVersioN
SUT	System Under Test
SPICE	Software Process Improvement and Capability Determination
TC	TestCase
TN	True Negative
TP	True Positive
UML	Unified Modeling Language
VW	VolksWagen
XML	eXtensible Markup Language

List of Figures

List of Tables

Bibliography

[1] Audi AG, *Self studie program audi a4, type 8w, 646*, Jul. 2015.

[2] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner, “Software engineering for automotive systems: A roadmap,” in *2007 Future of Software Engineering*, Washington, DC, USA: IEEE Computer Society, 2007, pp. 55–71, ISBN: 0-7695-2829-5. DOI: 10.1109/FOSE.2007.22. [Online]. Available: http://dx.doi.org/10.1109/FOSE.2007.22.

[3] Audi AG, *Self studie program audi a8, 459*, Nov. 2009.

[4] R. N. Charette, “This car runs on code,” *IEEE Spectrum*, vol. 46, no. 3, p. 3, 2009.

[5] D. Dvorak, “NASA study on flight software complexity,” American Institute of Aeronautics and Astronautics, Apr. 6, 2009, ISBN: 978-1-60086-979-2. DOI: 10.2514/6.2009-1882. [Online]. Available: http://arc.aiaa.org/doi/10.2514/6.2009-1882 (visited on 10/11/2016).

[6] M. Broy, “Mit welcher software faehrt das auto der zukunft,” vol. 06, pp. 92–97, 2011.

[7] ——, “Challenges in automotive software engineering,” in *Proceedings of the 28th international conference on Software engineering*, New York, NY, USA: ACM, 2006, pp. 33–42, ISBN: 1-59593-375-1. DOI: 10.1145/1134285.1134292. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134292.

[8] N. Aschenbrenner, “Ingenious electronics,” *Pictures of the Feature*, p. 55, 2005, ISSN: 1618-5498. [Online]. Available: https://www.siemens.com/content/dam/internet/siemens-com/innovation/pictures-of-the-future/pof-archive/pof-fall-2005.pdf (visited on 09/02/2016).

[9] M. Ehmer, *Automobiltechnologie 2010*, 2002. (visited on 09/02/2016).

[10] NHTSA, *Office of defects investigation (ODI) recalls database*, 1997. [Online]. Available: www-odi.nhtsa.dot.gov (visited on 03/31/2013).

[11] WARDS Auto, *U.s. car and truck sales, 1931-2013*, 2013. [Online]. Available: www.wardsauto.com (visited on 04/23/2014).

[12] J. Banks, L. Dixon, and J. Beckman, *The impact of vehicle recalls on the automotive market*, 2014. [Online]. Available: http://www.autonews.com/Assets/pdf/NADA%20UCG_WhitePaper_Impact%20of%20Vehicle%20Recalls.pdf (visited on 08/30/2016).

[13] H. Altinger, F. Wotawa, and M. Schurius, "Testing methods used in the automotive industry: Results from a survey," in *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*, San Jose, CA: ACM, Jul. 21, 2014, pp. 1–6. (visited on 07/21/2014).

[14] W. P. Klocwork, "Software on wheels," Klockwork.com, Oct. 2012, p. 6. [Online]. Available: www.klocwork.com (visited on 04/08/2014).

[15] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology, RTI Project*, vol. 7007, no. 11, 2002.

[16] J. Capers, *A short history of the cost per defect metric*, May 5, 2009. [Online]. Available: www.semat.org (visited on 04/08/2014).

[17] ISO, *ISO/IEC 14764:2006 software engineering – software life cycle processes – maintenance*. 2011.

[18] D. Kozlov, J. Koskinen, and M. Sakkinen, "Fault-proneness of open source software: Exploring its relations to internal software quality and maintenance process," *Open Software Engineering Journal*, vol. 7, pp. 1–23, 2013.

[19] G. Canfora, A. Cimitile, and P. B. Lucarelli, "Software maintenance," *Handbook of Software Eng. and Knowledge Eng*, pp. 91–120, 2002.

[20] F. Shull, V. Basili, B. Boehm, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, M. Zelkowitz, *et al.*, "What we have learned about fighting defects," in *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, 2002, pp. 249–258.

[21] L. Jin-Hua, L. Qiong, and L. Jing, "The w-model for testing software product lines," in *Computer Science and Computational Technology, 2008. ISCSCT'08. International Symposium on*, vol. 1, 2008, pp. 690–693.

[22] H. Altinger, F. Bock, F. Wotawa, and R. German, "Test and test-automation in the automotive industry," *SUBMITED to: IEEE Transactions on Software Engineering*, 2017.

[23] H. Altinger, S. Siegl, Y. Dajsuren, and F. Wotawa, "A novel industry grade dataset for fault prediction based on model-driven developed automotive embedded software," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, Florence, Italy: IEEE, May 16, 2015.

[24] H. Altinger, S. Herbold, J. Grabowski, and F. Wotawa, "Novel insights on cross project fault prediction applied to automotive software," in *Testing Software and Systems*, K. El-Fakih, G. Barlas, and N. Yevtushenko, Eds., vol. 9447, Springer International Publishing, 2015, pp. 141–157, ISBN: 978-3-319-25944-4. [Online]. Available: `http://dx.doi.org/10.1007/978-3-319-25945-1_9`.

[25] H. Altinger, S. Herbold, F. Wotawa, and F. Schneemann, "Performance tuning for automotive software fault prediction," in *2017 IEEE 24th International Conference on Software Analysis, Evolution, and Reengineering*, Klagenfurt, Austria: IEEE, Feb. 24, 2017.

[26] H. Altinger, Y. Dajsuren, S. Sieg, J. J. Vinju, and F. Wotawa, "On error-class distribution in automotive model-based software," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, Osaka, Japan: IEEE, Mar. 17, 2016, pp. 688–692, ISBN: 978-1-5090-1855-0. DOI: `10.1109`.

[27] H. Winner and G. Wolf, *Handbuch fahrerassistenzsysteme grundlagen, komponenten und systeme für aktive sicherheit und komfort*. Wiesbaden: Vieweg+Teubner Verlag / Springer Fachmedien Wiesbaden GmbH, Wiesbaden, 2012, ISBN: 978-3-8348-8619-4. [Online]. Available: `http://books.google.at/books?id=Lz1G7L7xgR0C`.

[28] H. Braess and U. Seiffert, *Vieweg handbuch kraftfahrzeugtechnik*, ser. ATZ/MTZ-Fachbuch. Springer Fachmedien Wiesbaden, 2013, ISBN: 978-3-658-01691-3.

[29] D. Crolla, *Encyclopedia of automotive engineering*. Wiley, 2015, ISBN: 978-0-470-97402-5. [Online]. Available: https://books.google.at/books?id=ANfdCgAAQBAJ.

[30] G. Volpato and A. Stocchetti, "Managing product life cycle in the auto industry: Evaluating carmakers effectiveness," *International Journal of Automotive Technology and Management*, vol. 8, no. 1, pp. 22–41, 2008.

[31] M. Broy, I. Kruger, A. Pretschner, and C. Salzmann, "Engineering automotive software," *Proceedings of the IEEE*, vol. 95, no. 2, pp. 356–373, 2007, ISSN: 0018-9219. DOI: 10.1109/JPROC.2006.888386.

[32] KBA, *Bestand nach ausgewählten fahrzeugklassen mit dem durchschnittsalter der fahrzeuge am 1. januar 2014*, 2014. [Online]. Available: http://www.kba.de/DE/Statistik/Fahrzeuge/Bestand/Fahrzeugalter/2014_b_fahrzeugalter_kfz_dusl.html (visited on 04/22/2015).

[33] D. Sabadka, "Impacts of shortening product life cycle in the automotive industry," *Transfer inovácií*, pp. 29–2013, 2013.

[34] A. S. G. Andrae and O. Andersen, "Life cycle assessments of consumer electronics — are they consistent?" *The International Journal of Life Cycle Assessment*, vol. 15, no. 8, pp. 827–836, 2010, ISSN: 1614-7502. DOI: 10.1007/s11367-010-0206-1. [Online]. Available: http://dx.doi.org/10.1007/s11367-010-0206-1.

[35] V. von Tils, "Design requirements for automotive reliability," in *Procedings of the ROBOSPICE Workshop*, Montreux, Sep. 22, 2016. [Online]. Available: https://goo.gl/5nOZBx (visited on 09/01/2016).

[36] B. Hardung, T. Kölzow, and A. Krüger, "Reuse of software in distributed embedded automotive systems," in *Proceedings of the 4th ACM international conference on Embedded software*, ACM, 2004, pp. 203–210.

[37] J. Peleska, F. Lapschies, H. Löding, P. Smuda, H. Schmid, E. Vorobev, and C. Zahlten, *Turn indicator model overview*, Apr. 18, 2014. [Online]. Available: http://www.informatik.uni-bremen.de/agbs/testingbenchmarks/turn_indicator/ (visited on 10/01/2014).

[38] F. Bock, D. Homm, S. Siegl, and R. German, "A taxonomy for tools, processes and languages in automotive software engineering," *CORR*, vol. abs/1601.03528, 2016. [Online]. Available: http://arxiv.org/abs/1601.03528.

[39] J. Schäuffele and T. Zurawka, *Automotive software engineering: Grundlagen, prozesse, methoden und werkzeuge effizient einsetzen*. Springer DE, 2013.

[40] P. Haberl, A. Spillner, K. Vosseberg, and M. Winter, "Survey 2011: Software test in practice," German testing board, 2011.

[41] The Mathworks, *Simulink*, 2015. [Online]. Available: http://de.mathworks.com/products/simulink (visited on 11/25/2015).

[42] dSpace, *TargetLink*, 2015. [Online]. Available: https://www.dspace.com/de/gmb/home/products/sw/pcgs/targetli.cfm (visited on 02/26/2015).

[43] Motor Industry Software Reliability Association, *Development guidelines for vehicle based software*. Warwickshire: MISRA, 1994, 88 pp., ISBN: 978-0-9524156-0-2.

[44] ——, *MISRA-c:2004 - guidelines for the use of the c language in critical systems*, 2nd ed. Warwickshire: MISRA, 2004, 116 pp., ISBN: 978-0-9524156-2-6.

[45] C. Boogerd and L. Moonen, "Assessing the value of coding standards: An empirical study," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, 2008, pp. 277–286.

[46] S. Dersten, J. Axelsson, and J. Froberg, "Effect analysis of the introduction of AUTOSAR: A systematic literature review," in *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, Aug. 2011, pp. 239–246. DOI: 10.1109/SEAA.2011.44.

[47] C. Autosar. (Mar. 4, 2014). Autosar, Autosar Standard, [Online]. Available: www.autosar.org.

[48] Automotive SIG, *Automotive SPICE*. The SPICE User Group, May 10, 2010, vol. 2.5, 146 pp. [Online]. Available: www.automotivespice.com.

[49] ISO TC 22 SC 3, *ISO 26262:2011:road vehicles – functional safety*. Geneva: International, 2011, 486 pp. [Online]. Available: www.iso.org.

[50] R. Bergmann and R. Walesch, "HiL strategie audi," in *6. dSpace Anwender Konfernz 2012*, Stuttgart, Germany: dSpace, Jan. 29, 2012. [Online]. Available: http://www.dspace.com/shared/data/pdf/ankon2013/tag1_pdf/2_audi_walesch_robert_bergmann_richard.pdf.

[51] S.-O. Mueller, M. Brand, S. Wachendorf, H. Schroeder, T. Szot, S. Schwab, and B. Kremer, "Integration vernetzter fahrerassistenz-funktionen mit HiL fuer den VW passat CC," *ATZEXTRA*, vol. 14, no. 4, pp. 60–65, 2009.

[52] D.-W.-I. R. Pfeffer and D.-I. S. Schmidt, "Seamless tool chain for testing camera-based advanced driver assistance systems," *ATZELEKTRONIK worldwide*, vol. 10, no. 6, pp. 48–53, 2015.

[53] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.

[54] M. H. Halstead, *Elements of software science (operating and programming systems series)*. New York, NY, USA: Elsevier Science Inc., 1977, ISBN: 0-444-00205-7.

[55] B. Curtis, S. B. Sheppard, and P. Milliman, "Third time charm: Stronger prediction of programmer performance by software complexity metrics," in *Proceedings of the 4th international conference on Software engineering*, 1979, pp. 356–360.

[56] V. R. Basili and T.-Y. Phillips, "Evaluating and comparing software metrics in the software engineering laboratory," *SIGMETRICS Perform. Eval. Rev.*, vol. 10, no. 1, pp. 95–106, Jan. 1981, ISSN: 0163-5999. DOI: 10.1145/1010627.807913. [Online]. Available: http://doi.acm.org/10.1145/1010627.807913.

[57] E. N. Adams, "Optimizing preventive service of software products," *IBM Journal of Research and Development*, vol. 28, no. 1, pp. 2–14, 1984.

[58] Y. S. Sherif, E. Ng, and J. Steinbacher, "Computer software quality measurements and metrics," *Microelectronics Reliability*, vol. 25, no. 6, pp. 1105–1150, 1985, ISSN: 0026-2714. DOI: http://dx.doi.org/10.1016/0026-2714(85)90486-X. [Online]. Available: http://www.sciencedirect.com/science/article/pii/002627148590486X.

[59] B. Boehm, *Industrial software metrics top 10 list*. IEEE COMPUTER SOC 10662 LOS VAQUEROS CIRCLE, PO BOX 3014, LOS ALAMITOS, CA 90720-1264, 1987.

[60] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994, ISSN: 0098-5589. DOI: 10.1109/32.295895.

[61] N. Fenton, “Software measurement: A necessary scientific basis,” *Software Engineering, IEEE Transactions on*, vol. 20, no. 3, pp. 199–206, 1994.

[62] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel, “Early quality prediction: A case study in telecommunications,” *IEEE Softw.*, vol. 13, no. 1, pp. 65–71, Jan. 1996, ISSN: 0740-7459. DOI: 10.1109/52.476287. [Online]. Available: http://dx.doi.org/10.1109/52.476287.

[63] T. M. Khoshgoftaar, E. B. Allen, J. P. Hudepohl, and S. J. Aud, “Application of neural networks to software quality modeling of a very large telecommunications system,” *IEEE Transactions on Neural Networks*, vol. 8, no. 4, pp. 902–909, Jul. 1997, ISSN: 1045-9227. DOI: 10.1109/72.595888.

[64] S. L. Pfleeger, R. Jeffery, B. Curtis, and B. Kitchenham, “Status report on software measurement,” *IEEE software*, vol. 14, no. 2, pp. 33–43, 1997.

[65] N. E. Fenton and M. Neil, “Software metrics: Successes, failures and new directions,” *Journal of Systems and Software*, vol. 47, no. 2, pp. 149–157, 1999.

[66] N. E. Fenton and N. Ohlsson, “Quantitative analysis of faults and failures in a complex software system,” *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 797–814, Aug. 2000, ISSN: 0098-5589. DOI: 10.1109/32.879815.

[67] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, “Predicting fault incidence using software change history,” *Software Engineering, IEEE Transactions on*, vol. 26, no. 7, pp. 653–661, 2000.

[68] X. Zhang and H. Pham, "An analysis of factors affecting software reliability," *Journal of Systems and Software*, vol. 50, no. 1, pp. 43–56, 2000.

[69] N. Japkowicz, "Learning from imbalanced data sets: A comparison of various strategies," in *AAAI workshop on learning from imbalanced data sets*, vol. 68, Menlo Park, CA, 2000, pp. 10–15.

[70] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Computer*, vol. 34, no. 1, pp. 135–137, Jan. 2001, ISSN: 0018-9162. DOI: 10.1109/2.962984. [Online]. Available: http://dx.doi.org/10.1109/2.962984.

[71] L. C. Briand, W. L. Melo, and J. Wust, "Assessing the applicability of fault-proneness models across object-oriented software projects," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 706–720, Jul. 2002, ISSN: 0098-5589. DOI: 10.1109/TSE.2002.1019484.

[72] G. Denaro and M. Pezze, "An empirical evaluation of fault-proneness models," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, May 2002, pp. 241–251. DOI: 10.1145/581368.581371.

[73] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," in *ACM SIGSOFT Software Engineering Notes*, vol. 27, 2002, pp. 55–64.

[74] T. Menzies, J. D. Stefano, K. Ammar, K. McGill, P. Callis, J. Davis, and R. Chapman, "When can we test less?" In *Software Metrics Symposium, 2003. Proceedings. Ninth International*, Sep. 2003, pp. 98–110. DOI: 10.1109/METRIC.2003.1232459.

[75] C. Drummond, R. C. Holte, *et al.*, "C4. 5, class imbalance, and cost sensitivity: Why under-sampling beats over-sampling," in *Workshop on learning from imbalanced datasets II*, vol. 11, Citeseer, 2003.

[76] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, Nov. 2004, pp. 417–428. DOI: 10.1109/ISSRE.2004.35.

[77] H. Do, S. Elbaum, and G. Rothermel, “Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact,” *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[78] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, “Predicting the location and number of faults in large software systems,” *Software Engineering, IEEE Transactions on*, vol. 31, no. 4, pp. 340–355, 2005.

[79] J. Sayyad Shirabad and T. Menzies, *The PROMISE repository of software engineering databases.* 2005, Published: School of Information Technology and Engineering, University of Ottawa, Canada. [Online]. Available: http://promise.site.uottawa.ca/SERepository.

[80] V. Vipindeep and P. Jalote, “List of common bugs and programming practices to avoid them,” *Electronic, March*, 2005.

[81] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger, “Comparing bug finding tools with reviews and tests,” in *Testing of Communicating Systems: 17th IFIP TC6/WG 6.1 International Conference, TestCom 2005, Montreal, Canada, May 31 - June, 2005. Proceedings*, F. Khendek and R. Dssouli, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 40–55, ISBN: 978-3-540-32076-0. [Online]. Available: http://dx.doi.org/10.1007/11430230_4.

[82] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, “Looking for bugs in all the right places,” in *Proceedings of the 2006 international symposium on Software testing and analysis*, 2006, pp. 61–72.

[83] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead, “Automatic identification of bug-introducing changes,” in *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, Tokyo, Japan: IEEE Computer Society, 2006, pp. 81–90, ISBN: 0-7695-2579-2. DOI: 10.1109/ASE.2006.23.

[84] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, “Have things changed now?: An empirical study of bug characteristics in modern open source software,” in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, 2006, pp. 25–33.

[85] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 452–461.

[86] P. Tomaszewski and L.-O. Damm, "Comparing the fault-proneness of new and modified code: An industrial case study," in *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ser. ISESE '06, New York, NY, USA: ACM, 2006, pp. 2–7, ISBN: 1-59593-218-6. DOI: 10.1145/1159733.1159737. [Online]. Available: http://doi.acm.org/10.1145/1159733.1159737.

[87] H. Krisp, K. Lamberg, and R. Leinfellner, "Automated real-time testing of electronic control units," SAE Technical Paper, 2007. [Online]. Available: papers.sae.org.

[88] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2–13, 2007.

[89] O. Mizuno and T. Kikuno, "Training on errors experiment to detect fault-prone software modules by spam filter," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07, New York, NY, USA: ACM, 2007, pp. 405–414, ISBN: 978-1-59593-811-4. DOI: 10.1145/1287624.1287683. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287683.

[90] T. J. Ostrand and E. J. Weyuker, "How to measure success of fault prediction models," in *Fourth international workshop on Software quality assurance: In conjunction with the 6th ESEC/FSE joint meeting*, 2007, pp. 25–30.

[91] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Using developer information as a factor for fault prediction," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, 2007, p. 8.

[92] M. Zhang and N. Baddoo, "Performance comparison of software complexity metrics in an open source project," in *Software Process Improvement*, Springer, 2007, pp. 160–174.

[93] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects for eclipse,” in *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, IEEE, 2007, pp. 9–9.

[94] I. Gondra, “Applying machine learning to software fault-proneness prediction,” *Journal of Systems and Software*, vol. 81, no. 2, pp. 186–195, 2008, ISSN: 0164-1212. DOI: http://dx.doi.org/10.1016/j.jss.2007.05.035. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121207001240.

[95] Y. Jiang, B. Cuki, T. Menzies, and N. Bartlow, “Comparing design and code metrics for software quality prediction,” in *Proceedings of the 4th international workshop on Predictor models in software engineering*, 2008, pp. 11–18.

[96] Y. Kamei, A. Monden, S. Morisaki, and K.-i. Matsumoto, “A hybrid faulty module prediction using association rule mining and logistic regression analysis,” in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '08, New York, NY, USA: ACM, 2008, pp. 279–281, ISBN: 978-1-59593-971-5. DOI: 10.1145/1414004.1414051. [Online]. Available: http://doi.acm.org/10.1145/1414004.1414051.

[97] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, “Benchmarking classification models for software defect prediction: A proposed framework and novel findings,” *Software Engineering, IEEE Transactions on*, vol. 34, no. 4, pp. 485–496, 2008.

[98] R. Lincke, J. Lundberg, and W. Löwe, “Comparing software metrics tools,” in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08, New York, NY, USA: ACM, 2008, pp. 131–142, ISBN: 978-1-60558-050-0. DOI: 10.1145/1390630.1390648. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390648.

[99] R. Moser, W. Pedrycz, and G. Succi, “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, 2008, pp. 181–190.

[100] O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. De Backer, and R. Haesen, "Mining software repositories for comprehensible software fault prediction models," *Journal of Systems and software*, vol. 81, no. 5, pp. 823–839, 2008.

[101] C. Boogerd and L. Moonen, "Evaluating the relation between coding standard violations and faultswithin and across software versions," in *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, 2009, pp. 41–50.

[102] C. Catal and B. Diri, "Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem," *Information Sciences*, vol. 179, no. 8, pp. 1040–1058, 2009.

[103] ——, "A systematic review of software fault prediction studies," *Expert systems with applications*, vol. 36, no. 4, pp. 7346–7354, 2009.

[104] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[105] I. Herraiz, D. Izquierdo-Cortazar, and F. Rivas-Hernández, "Flossmetrics: Free/libre/open source software metrics," in *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, IEEE, 2009, pp. 281–284.

[106] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, ser. PROMISE '09, New York, NY, USA: ACM, 2009, 7:1–7:10, ISBN: 978-1-60558-634-2. DOI: 10.1145/1540438.1540448. [Online]. Available: http://doi.acm.org/10.1145/1540438.1540448.

[107] A. Mockus, "Amassing and indexing a large sample of version control systems: Towards the census of public source code history.," in *MSR*, vol. 9, 2009, pp. 11–20.

[108] N. Schneidewind, *Systems and software engineering with applications*. New York NY: Institute of Electrical and Electronics Engineers, 2009. [Online]. Available: http://cds.cern.ch/record/1480945.

[109] Y. Singh, A. Kaur, and R. Malhotra, "Software fault proneness prediction using support vector machines," in *Proceedings of the world congress on engineering*, vol. 1, 2009, pp. 1–3.

[110] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Softw. Engg.*, vol. 14, no. 5, pp. 540–578, Oct. 2009, ISSN: 1382-3256. DOI: 10.1007/s10664-008-9103-7. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9103-7.

[111] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 91–100.

[112] A. Causevic, D. Sundmark, and S. Punnekkat, "An industrial survey on contemporary aspects of software testing," in *2010 Third International Conference on Software Testing, Verification and Validation*, Apr. 2010, pp. 393–401. DOI: 10.1109/ICST.2010.52.

[113] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, IEEE CS Press, 2010, pp. 31–41.

[114] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: Current results, limitations, new approaches," *Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, 2010, ISSN: 1573-7535. DOI: 10.1007/s10515-010-0069-5. [Online]. Available: http://dx.doi.org/10.1007/s10515-010-0069-5.

[115] O. Mizuno and H. Hata, "Prediction of fault-prone modules using a text filtering based metric," *International Journal of Software Engineering and Its Applications*, vol. 4, no. 1, 2010.

[116] T. M. Khoshgoftaar, K. Gao, and N. Seliya, "Attribute selection and imbalanced data: Problems in software defect prediction," in *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, vol. 1, Oct. 2010, pp. 137–144. DOI: 10.1109/ICTAI.2010.27.

[117] D. Posnett, V. Filkov, and P. Devanbu, "Ecological inference in empirical software engineering," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11, Washington, DC, USA: IEEE Computer Society, 2011, pp. 362–371, ISBN: 978-1-4577-1638-6. DOI: 10.1109/ASE.2011.6100074. [Online]. Available: http://dx.doi.org/10.1109/ASE.2011.6100074.

[118] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4, pp. 531–577, 2012.

[119] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang, "An investigation on the feasibility of cross-project defect prediction," *Automated Software Engineering*, vol. 19, no. 2, pp. 167–199, 2012, ISSN: 0928-8910. DOI: 10.1007/s10515-011-0090-3. [Online]. Available: http://dx.doi.org/10.1007/s10515-011-0090-3.

[120] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the "imprecision"of cross-project defect prediction," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12, New York, NY, USA: ACM, 2012, 61:1–61:11, ISBN: 978-1-4503-1614-9. DOI: 10.1145/2393596.2393669. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393669.

[121] C. Rommel and A. Girard, *Automated defect prevention for embedded software quality*, Feb. 2012. [Online]. Available: alm.parasoft.com (visited on 04/10/2014).

[122] S. Herbold, "Training data selection for cross-project defect prediction," in *Proceedings of the 9th International Conference on Predictive Models in Software Engineering*, ser. PROMISE '13, New York, NY, USA: ACM, 2013, 6:1–6:10, ISBN: 978-1-4503-2016-0. DOI: 10.1145/2499393.2499395. [Online]. Available: http://doi.acm.org/10.1145/2499393.2499395.

[123] W. P. MathWorks, "Comprehensive static analysis using polyspace products," MathWorks, Sep. 2013, p. 12. [Online]. Available: www.mathworks.com (visited on 03/25/2014).

[124] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 382–391.

[125] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Information and Software Technology*, vol. 55, no. 8, pp. 1397–1418, 2013. DOI: http://dx.doi.org/10.1016/j.infsof.2013.02.009.

[126] G. M. Weiss, "Foundations of imbalanced learning," in *Imbalanced Learning*, John Wiley & Sons, Inc., 2013, pp. 13–41, ISBN: 978-1-118-64610-6. [Online]. Available: http://dx.doi.org/10.1002/9781118646106.ch2.

[127] G. Abaei and A. Selamat, "A survey on software fault detection based on different prediction approaches," *Vietnam Journal of Computer Science*, vol. 1, no. 2, pp. 79–95, 2014, ISSN: 2196-8896. DOI: 10.1007/s40595-013-0008-z. [Online]. Available: http://dx.doi.org/10.1007/s40595-013-0008-z.

[128] R. Jus, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Defects4J: A database of existing faults to enable controlled testing studies for Java programs*, San Jose: ACM, Jun. 2014, pp. 437–440, ISBN: 978-1-4503-2645-2. DOI: http://dx.doi.org/10.1145/2610384.2628055. [Online]. Available: http://defects4j.org.

[129] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 182–191.

[130] P. He, B. Li, X. Liu, J. Chen, and Y. Ma, "An empirical study on software defect prediction with a simplified metric set," *Information and Software Technology*, vol. 59, pp. 170–190, 2015, ISSN: 0950-5849. DOI: http://dx.doi.org/10.1016/j.infsof.2014.11.006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584914002523.

[131] T. Menzies, R. Krishna, and D. Pryor, *The promise repository of empirical software engineering data*. North Carolina State University, 2015. [Online]. Available: http://openscience.us/repo.

[132] B. Klaus Rüdiger, "Marktanalyse zu sicherheitskritischer software in der automobilbranche," Masterthesis, Karlsruher Institut für Technologie, Karlsruhe, Apr. 22, 2016, 256 pp.

[133] NASA, *Metrics data program data repository*, 2004. [Online]. Available: http://mdp.ivv.nasa.gov.

[134] Y. Jiang, B. Cukic, and Y. Ma, "Techniques for evaluating fault prediction models," *Empirical Software Engineering*, vol. 13, no. 5, pp. 561–595, 2008, ISSN: 1573-7616. DOI: 10.1007/s10664-008-9079-3. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9079-3.

[135] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang, "Implications of ceiling effects in defect predictors," in *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, ser. PROMISE '08, New York, NY, USA: ACM, 2008, pp. 47–54, ISBN: 978-1-60558-036-4. DOI: 10.1145/1370788.1370801. [Online]. Available: http://doi.acm.org/10.1145/1370788.1370801.

[136] H. Altinger, *Survey on automated testing @ automotive domain*, Mar. 1, 2014. [Online]. Available: www.ist.tugraz.at/_attach/Publish/AltingerHarald/Survey_automated_testing_enu.pdf (visited on 03/01/2014).

[137] IBM, *Rational doors*, 2015. [Online]. Available: http://www-03.ibm.com/software/products/en/ratidoor (visited on 11/25/2015).

[138] PTC, *Integrity*, 2015. [Online]. Available: http://www.ptc.com/product/integrity/automotive (visited on 02/26/2015).

[139] Sparx Systems, *Enterprise architect*, 2015. [Online]. Available: http://www.sparxsystems.com/products/ea/ (visited on 11/25/2015).

[140] AEV, *EXtended automated conformance test*, 2015. [Online]. Available: http://www.audi-electronics-venture.de/aev/brand/en/services/development_tools/EXACT.html (visited on 11/25/2015).

[141] The Mathworks, *Polyspace*, 2015. [Online]. Available: http://de.mathworks.com/products/polyspace/ (visited on 11/25/2015).

[142] MicroNova, *EXtended automation method*, 2015. [Online]. Available: http://www.micronova.de/en/testing/testautomation/exam.html (visited on 11/25/2015).

[143] PikeTec, *Time partition testing*, 2015. [Online]. Available: https://www.piketec.com/en/2/tpt.html (visited on 11/25/2015).

[144] Vector Informatik, *CANOe*, 2015. [Online]. Available: http://vector.com/vi_canoe_en.html (visited on 11/25/2015).

[145] QA Systems, *QA-c*, 2015. [Online]. Available: http://www.qa-systems.de/produkte/qa-c.html (visited on 11/25/2015).

[146] G. Kiffe and T. Bock, “Standardisierte entwicklungsumgebung für die softwareeigenentwicklung bei audi,” *DSPACE Magazine*, vol. 1/2013, Jan. 29, 2013. [Online]. Available: https://www.dspace.com/shared/data/pdf/ankon2013/Tag2_PDF/2_Audi_Kiffe-Gerhard_Bock-Thomas.pdf (visited on 05/15/2015).

[147] T. Mathworks, *Mathworks automotive advisory board checks (MAAB)*, 2014. [Online]. Available: http://de.mathworks.com/help/slvnv/ref/mathworks-automotive-advisory-board-checks.html (visited on 02/26/2014).

[148] Model Engineering Solution, *Model examiner*, 2015. [Online]. Available: http://www.model-engineers.com/en/model-examiner.html (visited on 11/25/2015).

[149] H. Altinger, *Dataset on automotive software repository*, Feb. 26, 2015. [Online]. Available: http://www.ist.tugraz.at/_attach/Publish/AltingerHarald/MSR_2015_dataset_automotive.zip (visited on 03/01/2014).

[150] The Mathworks, *Stateflow*, 2015. [Online]. Available: http://de.mathworks.com/products/stateflow (visited on 11/25/2015).

[151] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, “Code quality analysis in open source software development,” *Information Systems Journal*, vol. 12, no. 1, pp. 43–60, 2002.

[152] locmetrics.com, *LocMetric*, 2015. [Online]. Available: http://www.locmetrics.com/ (visited on 02/26/2015).

[153] H. Israel, *CMetric*, 2015. [Online]. Available: https://github.com/MetricsGrimoire/CMetrics (visited on 02/26/2015).

[154] Scooter Software, *Beyond compare 4*, 2015. [Online]. Available: http://www.scootersoftware.com/ (visited on 02/26/2015).

[155] MySQL, *MySQL database server*, 2015. [Online]. Available: http://www.mysql.com/ (visited on 02/26/2015).

[156] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *Journal of Machine learning research*, vol. 7, pp. 1–30, Jan 2006.

[157] S. S. Shapiro and R. S. Francia, "An approximate analysis of variance test for normality," *Journal of the American Statistical Association*, vol. 67, no. 337, pp. 215–216, 1972. DOI: 10.1080/01621459.1972.10481232. [Online]. Available: http://amstat.tandfonline.com/doi/abs/10.1080/01621459.1972.10481232.

[158] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[159] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum, "Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 402–419, Jun. 2007, ISSN: 0098-5589. DOI: 10.1109/TSE.2007.1015.

[160] M. G. Kendall, "Rank correlation methods.," 1948.

[161] K. Pearson, *On the theory of contingency and its relation to association and normal correlation*, ser. Drapers Company Research Memories Biometric Series I. London: Dulau and Co., 1904, vol. Mathematical contributions to the theory of Evolution, 40 pp.

[162] C. Spearman, ""General intelligence", objectively determined and measured," *The American Journal of Psychology*, vol. 15, no. 2, pp. 201–292, 1904.

[163] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *Software Engineering, IEEE Transactions on*, vol. 25, no. 5, pp. 675–689, 1999.

[164] R. Rana, M. Staron, J. Hansson, and M. Nilsson, "Defect prediction over software life cycle in automotive domain," in *Proceedings of the International Joint conference on Software Technologies 2014*, Vienna, Austria, Aug. 29, 2014.

[165] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *2011 33rd International Conference on Software Engineering (ICSE)*, May 2011, pp. 481–490. DOI: 10.1145/1985793.1985859.

[166] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15, Piscataway, NJ, USA: IEEE Press, 2015, pp. 99–108. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819026.

[167] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms," in *Proc. of KDD-2013*, 2013, pp. 847–855.

[168] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[169] C.-J. Hsieh, S. Si, and I. S. Dhillon, "A divide-and-conquer solver for kernel support vector machines," in *Proceedings of the 31st International Conference on Machine Learning*, vol. 32, Beijing, China: JMLR, 2014. [Online]. Available: http://arxiv.org/abs/1311.0914.

[170] T. Therneau, E. Atkinson, and M. Foundation, *An introduction to recursive partitioning using the RPART routines*, Jun. 29, 2015. [Online]. Available: https://cran.r-project.org/web/packages/rpart/vignettes/longintro.pdf.

[171] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," *CORR*, vol. abs/1603.02754, 2016. [Online]. Available: http://arxiv.org/abs/1603.02754.

[172] D. E. Perry, A. A. Porter, and L. G. Votta, "Empirical studies of software engineering: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00, New York, NY, USA: ACM, 2000, pp. 345–355, ISBN: 1-58113-253-0. DOI: 10.1145/336512.336586. [Online]. Available: http://doi.acm.org/10.1145/336512.336586.

[173] X.-B. D. Le, L. David, and C. L. Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, Osaka, Japan: IEEE, Mar. 17, 2016, pp. 213–224, ISBN: 978-1-5090-1855-0. DOI: 10.1109.

[174] H. Altinger, S. Galler, S. Mühlbacher-Karrer, G. Steinbauer, F. Wotawa, and H. Zangl, "Concept evaluation of a reflex inspired ball handling device for autonomous soccer robots," in *ROBOCUP 2009: Robot Soccer World Cup XIII*, J. Baltes, M. Lagoudakis, T. Naruse, and S. Ghidary, Eds., vol. 5949, Springer Berlin Heidelberg, 2010, pp. 11–22, ISBN: 978-3-642-11875-3. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11876-0_2.

[175] M. Kollar, H. Altinger, and M. Bader, "The application of pneumatic actuators in RoboCups' middle size league," in *The application of pneumatic actuators in RoboCups' Middle Size League*, Michael Hofbaur, Manfred Husty (Eds.), ., 2011, pp. 117–126, ISBN: 978-3-9503191-0-1.

[176] H. Altinger, "Ultrasonic sensor based self localisation for autonomous vehicles within mapped environments," Diplomathesis, Graz, University of Technology; TU Graz, Graz, Mar. 19, 2013, 124 pp.

[177] H. Altinger, F. Schuller, B. Müller-Beßler, and A. Reich, "Pilotiertes parken im parkhaus - möglichkeiten aus einem neuen ansatz durch die verbindung von infrastruktur und fahrzeug," in *VDI Gemeinschaftstagung - Elektronik im Fahrzeug*, vol. 2188, Baden Baden: VDI, Oct. 16, 2013, p. 784, ISBN: 978-3-18-092188-4. [Online]. Available: http://shop.vdi-nachrichten.com/buchshop/literaturshop/langanzeige.asp?vr_id=8540.

[178] A. Ibisch, S. Stümper, H. Altinger, M. Neuhausen, M. Tschentscher, M. Schlipsing, J. Salinen, and A. Knoll, "Towards autonomous driving in a parking garage: Vehicle localization and tracking using environment-embedded LIDAR sensors," in *Intelligent Vehicles Symposium (IV), 2013 IEEE*, IEEE, 2013, pp. 829–834.

[179] A. Ibisch, S. Stümper, H. Altinger, m. Neuhausen, M. Tschentscher, M. Schlipsing, J. Salmen, and A. Knoll, "Autonomous driving in a parking garage: Vehicle-localization and tracking using environment-embedded LIDAR sensors," presented at the Intelligent Vehicles

Symposium (IV), 2013 IEEE, Gold Coast: IEEE, 2013, pp. 829–834. DOI: 10.1109/IVS.2013.6629569.

[180] A. Ibisch, S. Houben, M. Schlipsing, R. Kesten, P. Reimche, F. Schuller, and H. Altinger, "Towards highly automated driving in a parking garage: General object localization and tracking using an environment-embedded camera system," in *2014 IEEE Intelligent Vehicles Symposium Proceedings*, IEEE, 2014, pp. 426–431.

[181] C. Sippl, F. Bock, D. Wittmann, H. Altinger, and R. German, "From simulation data to test cases for fully automated driving and ADAS," in *Testing Software and Systems: 28th IFIP WG 6.1 International Conference, ICTSS 2016, Graz, Austria, October 17-19, 2016, Proceedings*, F. Wotawa, M. Nica, and N. Kushik, Eds., Cham: Springer International Publishing, 2016, pp. 191–206, ISBN: 978-3-319-47443-4. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-47443-4_12.

[182] M. Olszewska, Y. Dajsuren, H. Altinger, A. Serebrenik, M. Waldén, and M. G. van den Brand, "Tailoring complexity metrics for simulink models," in *Proceedings of ECSA 2016*, Kopenhaben: Springer, Nov. 28, 2016. [Online]. Available: http://ecsa2016.icmc.usp.br/.

[183] P. Kruchten, Y. Dajsuren, H. Altinger, and M. Staron, "Proceedings of the first international workshop on automotive software architecture (WASA'15, montreal, canada, may 4, 2015)," 2015.

[184] "Method for operating an automatically driven, driverless motor vehicle and monitoring system," DE102014015075A1, WO Patent App. PCT/EP2015/001,967, Jul. 2016. [Online]. Available: https://www.google.com/patents/WO2016055159A3?cl=en.

[185] "Method for determining the absolute postion of a mobile unit, and mobile unit," DE DE20141002150 A1, DE Patent 102,014,013,208, Jan. 2016. [Online]. Available: http://google.co.il/patents/DE102014013208B3?cl=ko.

[186] H. Altinger, "Automotive software engineering and application of fault prediction," Presentation, Software Quality Symphosium 2016, Tokio, Japan, Sep. 19, 2016.

Third Party Tools

[41] The Mathworks, *Simulink*, 2015. [Online]. Available: http://de.mathworks.com/products/simulink (visited on 11/25/2015).

[42] dSpace, *TargetLink*, 2015. [Online]. Available: https://www.dspace.com/de/gmb/home/products/sw/pcgs/targetli.cfm (visited on 02/26/2015).

[137] IBM, *Rational doors*, 2015. [Online]. Available: http://www-03.ibm.com/software/products/en/ratidoor (visited on 11/25/2015).

[138] PTC, *Integrity*, 2015. [Online]. Available: http://www.ptc.com/product/integrity/automotive (visited on 02/26/2015).

[139] Sparx Systems, *Enterprise architect*, 2015. [Online]. Available: http://www.sparxsystems.com/products/ea/ (visited on 11/25/2015).

[140] AEV, *EXtended automated conformance test*, 2015. [Online]. Available: http://www.audi-electronics-venture.de/aev/brand/en/services/development_tools/EXACT.html (visited on 11/25/2015).

[141] The Mathworks, *Polyspace*, 2015. [Online]. Available: http://de.mathworks.com/products/polyspace/ (visited on 11/25/2015).

[142] MicroNova, *EXtended automation method*, 2015. [Online]. Available: http://www.micronova.de/en/testing/testautomation/exam.html (visited on 11/25/2015).

[143] PikeTec, *Time partition testing*, 2015. [Online]. Available: https://www.piketec.com/en/2/tpt.html (visited on 11/25/2015).

[144] Vector Informatik, *CANOe*, 2015. [Online]. Available: http://vector.com/vi_canoe_en.html (visited on 11/25/2015).

[145] QA Systems, *QA-c*, 2015. [Online]. Available: http://www.qa-systems.de/produkte/qa-c.html (visited on 11/25/2015).

[148] Model Engineering Solution, *Model examiner*, 2015. [Online]. Available: http://www.model-engineers.com/en/model-examiner.html (visited on 11/25/2015).

[152] locmetrics.com, *LocMetric*, 2015. [Online]. Available: http://www.locmetrics.com/ (visited on 02/26/2015).

[153] H. Israel, *CMetric*, 2015. [Online]. Available: https://github.com/MetricsGrimoire/CMetrics (visited on 02/26/2015).

[154] Scooter Software, *Beyond compare 4*, 2015. [Online]. Available: http://www.scootersoftware.com/ (visited on 02/26/2015).

[155] MySQL, *MySQL database server*, 2015. [Online]. Available: http://www.mysql.com/ (visited on 02/26/2015).

Altingers Publications

[13] H. Altinger, F. Wotawa, and M. Schurius, “Testing methods used in the automotive industry: Results from a survey,” in *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*, San Jose, CA: ACM, Jul. 21, 2014, pp. 1–6. (visited on 07/21/2014).

[23] H. Altinger, S. Siegl, Y. Dajsuren, and F. Wotawa, “A novel industry grade dataset for fault prediction based on model-driven developed automotive embedded software,” in *Proceedings of the 12th Working Conference on Mining Software Repositories*, Florence, Italy: IEEE, May 16, 2015.

[24] H. Altinger, S. Herbold, J. Grabowski, and F. Wotawa, “Novel insights on cross project fault prediction applied to automotive software,” in *Testing Software and Systems*, K. El-Fakih, G. Barlas, and N. Yevtushenko, Eds., vol. 9447, Springer International Publishing, 2015, pp. 141–157, ISBN: 978-3-319-25944-4. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-25945-1_9.

[26] H. Altinger, Y. Dajsuren, S. Sieg, J. J. Vinju, and F. Wotawa, “On error-class distribution in automotive model-based software,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, Osaka, Japan: IEEE, Mar. 17, 2016, pp. 688–692, ISBN: 978-1-5090-1855-0. DOI: 10.1109.

[136] H. Altinger, *Survey on automated testing @ automotive domain*, Mar. 1, 2014. [Online]. Available: www.ist.tugraz.at/_attach/Publish/AltingerHarald/Survey_automated_testing_enu.pdf (visited on 03/01/2014).

[149] ——, *Dataset on automotive software repository*, Feb. 26, 2015. [Online]. Available: http://www.ist.tugraz.at/_attach/Publish/

AltingerHarald/MSR_2015_dataset_automotive.zip (visited on 03/01/2014).

[174] H. Altinger, S. Galler, S. Mühlbacher-Karrer, G. Steinbauer, F. Wotawa, and H. Zangl, "Concept evaluation of a reflex inspired ball handling device for autonomous soccer robots," in *ROBOCUP 2009: Robot Soccer World Cup XIII*, J. Baltes, M. Lagoudakis, T. Naruse, and S. Ghidary, Eds., vol. 5949, Springer Berlin Heidelberg, 2010, pp. 11–22, ISBN: 978-3-642-11875-3. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11876-0_2.

[176] H. Altinger, "Ultrasonic sensor based self localisation for autonomous vehicles within mapped environments," Diplomathesis, Graz, University of Technology; TU Graz, Graz, Mar. 19, 2013, 124 pp.

[177] H. Altinger, F. Schuller, B. Müller-Beßler, and A. Reich, "Pilotiertes parken im parkhaus - möglichkeiten aus einem neuen ansatz durch die verbindung von infrastruktur und fahrzeug," in *VDI Gemeinschaftstagung - Elektronik im Fahrzeug*, vol. 2188, Baden Baden: VDI, Oct. 16, 2013, p. 784, ISBN: 978-3-18-092188-4. [Online]. Available: http://shop.vdi-nachrichten.com/buchshop/literaturshop/langanzeige.asp?vr_id=8540.

[178] A. Ibisch, S. Stümper, H. Altinger, M. Neuhausen, M. Tschentscher, M. Schlipsing, J. Salinen, and A. Knoll, "Towards autonomous driving in a parking garage: Vehicle localization and tracking using environment-embedded LIDAR sensors," in *Intelligent Vehicles Symposium (IV), 2013 IEEE*, IEEE, 2013, pp. 829–834.

[179] A. Ibisch, S. Stümper, H. Altinger, m. Neuhausen, M. Tschentscher, M. Schlipsing, J. Salmen, and A. Knoll, "Autonomous driving in a parking garage: Vehicle-localization and tracking using environment-embedded LIDAR sensors," presented at the Intelligent Vehicles Symposium (IV), 2013 IEEE, Gold Coast: IEEE, 2013, pp. 829–834. DOI: 10.1109/IVS.2013.6629569.

[180] A. Ibisch, S. Houben, M. Schlipsing, R. Kesten, P. Reimche, F. Schuller, and H. Altinger, "Towards highly automated driving in a parking garage: General object localization and tracking using an environment-embedded camera system," in *2014 IEEE Intelligent Vehicles Symposium Proceedings*, IEEE, 2014, pp. 426–431.

[181] C. Sippl, F. Bock, D. Wittmann, H. Altinger, and R. German, “From simulation data to test cases for fully automated driving and ADAS,” in *Testing Software and Systems: 28th IFIP WG 6.1 International Conference, ICTSS 2016, Graz, Austria, October 17-19, 2016, Proceedings*, F. Wotawa, M. Nica, and N. Kushik, Eds., Cham: Springer International Publishing, 2016, pp. 191–206, ISBN: 978-3-319-47443-4. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-47443-4_12.

[182] M. Olszewska, Y. Dajsuren, H. Altinger, A. Serebrenik, M. Waldén, and M. G. van den Brand, “Tailoring complexity metrics for simulink models,” in *Proceedings of ECSA 2016*, Kopenhaben: Springer, Nov. 28, 2016. [Online]. Available: http://ecsa2016.icmc.usp.br/.

[183] P. Kruchten, Y. Dajsuren, H. Altinger, and M. Staron, “Proceedings of the first international workshop on automotive software architecture (WASA’15, montreal, canada, may 4, 2015),” 2015.

Altingers Work submitted to review

[22] H. Altinger, F. Bock, F. Wotawa, and R. German, “Test and test-automation in the automotive industry,” *SUBMITED to: IEEE Transactions on Software Engineering*, 2017.

Altingers Patents

[184] "Method for operating an automatically driven, driverless motor vehicle and monitoring system," DE102014015075A1, WO Patent App. PCT/EP2015/001,967, Jul. 2016. [Online]. Available: https://www.google.com/patents/WO2016055159A3?cl=en.

[185] "Method for determining the absolute postion of a mobile unit, and mobile unit," DE DE20141002150 A1, DE Patent 102,014,013,208, Jan. 2016. [Online]. Available: http://google.co.il/patents/DE102014013208B3?cl=ko.