CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Prediction of Software Faults Based on Requirements and Design Interrelationships

Master's Thesis in Software Engineering

BASHAR NASSAR

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

Master's thesis 2016

Prediction of Software Faults Based on Requirements and Design Interrelationships

BASHAR NASSAR

Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2016

Prediction of Software Faults Based on Requirements and Design Interrelationships
BASHAR NASSAR

Supervisor: Riccardo Scandariato, Chalmers & University of Gothenburg
Supervisor: Ali Shahrokni, Systemite AB
Examiner: Eric Knauss, Computer Science and Engineering

Master's Thesis 2016
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2016

Prediction of Software Faults Based on Requirements and Design Interrelationships
BASHAR NASSAR
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Abstract

Traceability information between different system artifacts, like the one between requirements, architectural elements and the results of test cases can be used to expose interesting relationships between the early phases of the software development process and the software faults in the end product. For instance, complex dependencies between features and software components could lead to an increased level of flaws in the code. Such patterns can be detected and visualized as early warnings to the relevant stakeholders (e.g., the architect or the project manager). Ultimately, a fully-fledged prediction model can be developed if enough historical information is available from previous software projects. In this thesis work we investigate the relationships between the system design attributes and test case results looking for fault patterns using the traceability data and design metrics. Our intention is to use these patterns to predict the system faults in early stage of the development process. The ultimate goal is to introduce a method for building a decision support system based on historic product data. The research presented in this thesis is based on a quantitative case study conducted together with our industrial partner Systemite AB, where the raw data was provided by three Swedish automotive companies. We found that design attributes, such as a number of component in port or functionality could have a strong relation with the probability of failed tests, thus they could by use as an indicator to predict faults. Those faults could be avoided during the design phase, which will lead to improve the quality and reliability of the system and reduce its cost and development time.

Keywords: Software Fault Prediction, Early Stage of Development Life Cycle, Metrics, Requirement, Software Architecture, System Design, Design attributes, Traceability Links, System Quality.

Acknowledgements

I, the author, would like to express my gratitude to the following people for their participation in this project.
Ali Shahrokni, supervisor at Systemite AB
For guidance, optimism, and the help he has provided.
Riccardo Scandariato, supervisor at Chalmers
For his direct answers to questions and for sharing his academical expertise.
The Holy Systemite AB Team
For the good hospitality, support, and help.

Bashar Nassar, Gothenburg, May 2016

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Research Question & Hypotheses . . . 3
  - 1.2.1 Research Question . . . 3
  - 1.2.2 Hypotheses . . . 3
- 1.3 Thesis Outline . . . 4

2 Related Work **7**
- 2.1 Literature Collection . . . 7
- 2.2 Prediction of Fault at Implementation Phase . . . 8
- 2.3 Prediction of Fault at Early Phase . . . 9
  - 2.3.1 Design Metrics . . . 9
- 2.4 Traceability . . . 10
- 2.5 Software Fault Prediction Model . . . 11

3 Methodology **13**
- 3.1 Methodology . . . 13
- 3.2 Research Methodology Steps . . . 14
- 3.3 Limitation & Validity Threats . . . 16
  - 3.3.1 Internal Validity . . . 16
  - 3.3.2 External Validity . . . 16
  - 3.3.3 Conclusion Validity . . . 17
  - 3.3.4 Construct Validity . . . 17

4 Metrics **19**
- 4.1 System Metrics . . . 19
  - 4.1.1 Cyclomatic Complexity . . . 19
  - 4.1.2 Weight . . . 20
  - 4.1.3 Size . . . 20
- 4.2 Component Metrics . . . 21
  - 4.2.1 Interaction . . . 21
    - 4.2.1.1 Component Interaction . . . 21
    - 4.2.1.2 Actual Interactions . . . 21

4.2.1.3 Total Interactions Performed 22
4.2.2 Weight 22
4.3 Requirements Metrics 22
4.3.1 Sub-Requirements per Requirement 23
4.3.2 Number of Linked Components 23
4.4 Testing Metrics 23
4.4.1 Number of Test Case 23
4.4.2 Number of Passed Test Case 24
4.4.3 Number of Failed Test Case 24
4.5 Combination Metrics 24

5 Implementation 25
5.1 Systemite AB 25
5.1.1 SystemWeaver's Architecture 26
5.1.2 SystemWeaver's Models 28
5.2 Xpath Expression 29
5.3 SmartTrace 31
5.3.1 Implementation Validation 35

6 Results 37
6.1 Data Collection 37
6.2 Descriptive Statistics 38
6.2.1 Normality Test 38
6.2.2 Linearity Test 38
6.2.3 Spearman's Correlation 39
6.2.3.1 Correlation at System level 40
6.2.3.2 Correlation at Component Level 41
6.2.3.3 Correlation at Requirement Level 42
6.2.4 Mean Difference 43
6.3 Inferential Statistics 44
6.3.1 Mann–Whitney U Test 44
6.4 Data Analysis 47

7 Conclusion 51
7.1 Summary 51
7.2 Discussion 52
7.3 Future Works 54
7.3.1 Study Other Industries 54
7.3.2 Define More Metrics 54
7.3.3 System Changes and Faults 55
7.3.4 Improve SmartTrace 56

Bibliography 59

A Appendix 1 I

B Appendix 2 III

List of Figures

1.1 Relative cost to fix software defects per life cycle phase 1
1.2 Origin of software defects 2

3.1 Research methodology steps. 14

5.1 SystemWeaver's conceptual architecture meta model. 26
5.2 SystemWeaver's testing and verification meta model. 27
5.3 SystemWeaver's traceability between system artifacts, abstraction level, and versioning 28
5.4 SystemWeaver's integrated embedded systems development process. . 29
5.5 Metrics implemented in SystemWeaver. 30
5.6 Grid view for weight metric in SystemWeaver. 30
5.7 Interaction between SystemWeaver and SmartTrace. 32
5.8 SystemWeaver and MathNet APIs. 33
5.9 Test case results statistic. 34
5.10 Tool implementation for component metrics. 34
5.11 Tool implementation for the correlation between metrics and test case results 35

6.1 Histogram of in ports and out ports metrics data 38
6.2 Scatter Plot for the total interaction performed metrics and the percentage of failed test cases. 39
6.3 Test case results changes over versions 40
6.4 Test case results changes over time 41
6.5 The in port mean difference between the healthy and defected component. 44
6.6 Mann Whitney U test between two independent groups(healthy(0) and defect(1) component) and component metrics 45
6.7 Hypothesis test summary for the two group(healthy(0) and defect(1) component metrics 46
6.8 Mann Whitney U test between two independent groups(healthy(0) and defect(1) component) and requirement metrics 47
6.9 Hypothesis test summary for the two group(healthy(0) and defect(1) requirement metrics 47

7.1 The relationship between system changes and generate faults. 55

7.2 The SmartTrace fault prediction model using the traceability data in the early stage of development life cycle. 56

List of Tables

4.1 System weight metrics. 20
4.2 Component weight metrics. 22

6.1 Correlation degree categories 40
6.2 Spearman's rank correlation coefficient between percentage of failed test cases and component metrics. 42
6.3 Summary of the Spearman's correlation exists between component metrics and the percentage of failed test cases over 11 system versions. 42
6.4 Spearman's rank correlation coefficient between percentage of failed test cases and requirement metrics 43
6.5 Summary of the Spearman's correlation exists between requirement metrics and the percentage of failed test cases over 12 system versions 43

1
Introduction

This chapter shortly presents the existing state of software fault prediction in different stage of system development, and the important of predicting fault in the early stage of the development life cycle. The chapter further outlines the overall research question and hypothesis and how this report aims to convey the study's most important features.

1.1 Background

Software fault prediction has been intensively studied in academia and industry. Many researchers have tried to predict faults based on implementation-level parameters and source code metrics such as size of code, code change history, source code complexity, complexity of the implementation processes, and programming languages. One such study has investigated the effect of class size on fault-proneness [13]. Another study goes farther to investigate the relation between the complexity of source code modification and the faults prediction [9], therefore they conclude that high complex changes to a file will lead to higher chance of containing faults.

Having an early predictors (i.e., before the start of implementation and testing phases) is useful and desirable as it can reduce the cost of software development, as known from the cost models for fixing defects, the cost dramatically increase through the development process, e.g., by 6.5 times when moving from the design to the implementation phase. Similarly, it increases from 6.5 to 15 times, when moving from implementation to testing phase, and from 15 to 100 times by moving from testing to maintenance [16] and [17], as shown in Figure1.1.

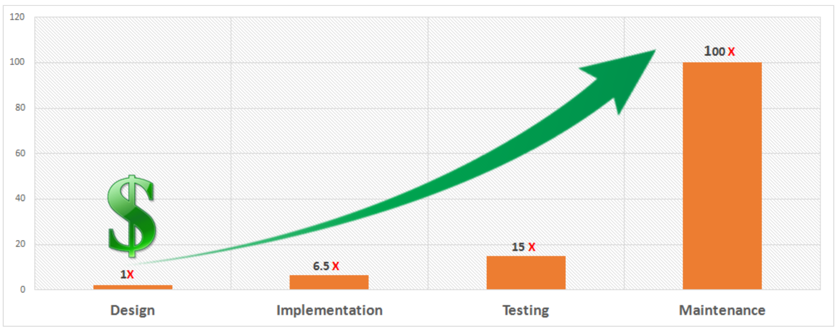


Figure 1.1: Relative cost to fix software defects per life cycle phase
Source: IBM Systems Science Institute.

Moreover, Mohan et al. show that 64% of defects are generated by the requirements analysis and design phases, while only 36 % of the defects are introduced in other development phase [16], as shown in Figure1.2.

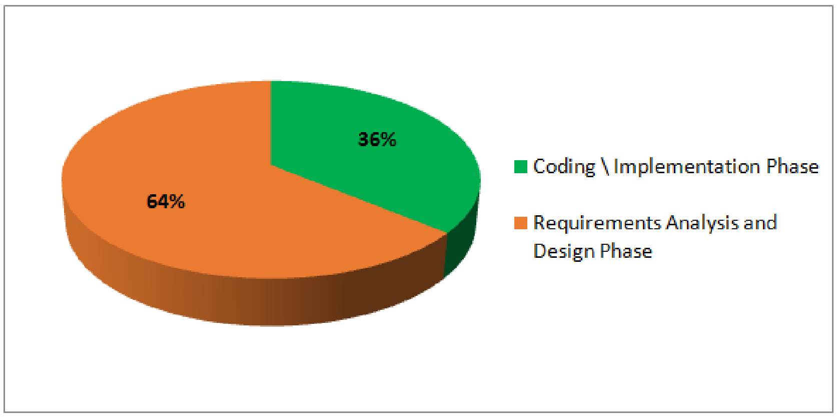


Figure 1.2: Origin of software defects
Source: Crosstalk, the journal of defense software engineering.

Fewer research studies exist with respect to the use of early defect predictor. For instance, a model using object oriented design metrics has been built to predict the faults during design phase. This model uses a technique based on neural networks and Bayesian Regularization (BR) algorithms [14]. Moreover, other works exist about using design properties as predictors. For instance, Rathore and Gupta [11] investigated the relationship between the class level object-oriented metrics with fault proneness of object-oriented software system. The results of their work showed that design attributes like coupling, complexity and size are correlated to fault proneness.

As a novel contribution, we plan on using information from requirements and design artifacts (and the traceability links between them) to predict faults[1]. In order to accomplish our goal and investigate our hypotheses we selected an application from the automotive industry. The hosting company was Systemite AB [44], and the raw data used to validates our research question and hypotheses comes from two Swedish automotive companies, the first company provides us with two mature systems and the second company provides one mature system, the name of these companies will be anonymous for the confidentiality of their data. The research present in this thesis work is based on a quantitative industrial case study conducted together with our partner Systemite.

The studied data comes from the automotive industry, it contains several mature projects that have been deployed and revised over time. The requirements and design for these projects are contained in SystemWeaver [43], an information management platform from Systemite [44]. The requirements are in rich text format with different textual styles and pictures, links, and sometimes more formal such as state machines and use cases. When it comes to design, SystemWeaver contains the architecture and design information at component- and signal-level and how signals

[1]In our work, we use the failures of test cases as a proxy for actual software faults.

propagate and components are connected. Also we can access both hardware and logical design (functions and features) and how they are allocated on to the hardware. Moving to testing part, the test is structure in test suits, every test suit is a collection of test cases that are intended to test a behavior or a set of behavior of a system. Moving down to test case level, test cases are related to a system or component under test and they often have traceability to individual requirements. Finally, the result of test cases is stored, and it could be passed, failed, or not complete.

The process of our work had divided mainly into four phases, which divided later into ten steps that form our research methodology. The result of each phase was an entrance for next one. The goal of phase one was to study the available data set, organized them, and eliminate inappropriate data. In phase two we define a set of design metrics, based on literature review for related work, Systemite's previous project and research, and the properties of the studied data. Moving to phase three we built a tool, which allow us to apply the design metrics that we have, investigate the correlation between metrics results and the test case result, and define the fault prediction pattern. Finally, in phase four we improved our tool in order to predict the system faults based on the historical product data.

1.2 Research Question & Hypotheses

1.2.1 Research Question

Our overall research question is whether the relationships between requirements and design artifacts (as well as information from individual artifacts) can be leveraged to predict software faults?

1.2.2 Hypotheses

In this thesis, we aim to validates the following six hypotheses:

1. A higher- complexity system design has a higher failure rate.
2. Software components that have a higher interaction with other component have a higher chance of failing a test.
3. Software components that have a higher number of responsibility have a higher chance of failing a test.
4. Software components that are responsible for (i.e., linked to) a higher number of requirements have a higher chance of failing a test.
5. Software requirement linked to large number of components tend to have more test failures than average.
6. Software requirement that have a higher number of sub-requirements have a higher chance of failing a test.

1.3 Thesis Outline

This introduction is followed by six other chapters, each of which covers a particular focus to the study, together with two appendixes that include our related publication.

CHAPTER 2 presents related works. The chapter begins by introducing the method used for selecting relevant literature, then it describes the fault prediction at different stages of development process, which are the implementation and design phases. Thereafter, design metrics and its uses for predicting fault are presented. Furthermore, traceability between system artifices described. Finally, software fault prediction models are presented.

CHAPTER 3 presents methodology. The research methodology of this thesis is quantitative case study conducted together with our industrial partner Systemite AB, and the raw data was provided by two Swedish automotive companies, that use Systemite's platform (SystemWeaver) to store and manage their data. The chapter organized as following, Section 3.1 presents the over all methodology. Section 3.2 further describes the research methodology steps. Finally, Section 3.3 discusses the research limitation and validity threats.

CHAPTER 4 presents design metrics. The design metrics are implemented in our work, in order to measure the design attributes, to be use later to investigate the relation between design attributes and system fault. The chapter organized as following, section 4.1 presents system metrics. Section 4.2 present component metrics. Section 4.3 present requirement metrics. Section 4.4 presents testing metric. Finally, Section 4.5 presents combined metrics.

CHAPTER 5 describes tool implementation. The development work could be divided into two stages, preparation for the study using Xpath expression and implementation of the study using C# WPF. The chapter begin with describing the hosting company and their platform. Then, it describes the using of Xpath expression to define design metrics and filtering our raw data. Finally, it presents the tool implementation, which has been used to extract the data and build our data set, we called it SmartTrace.

CHAPTER 6 presents results. This chapter summarizes the outcome of performing the case study, by presenting and interpreting the measurement result, which is crucial for understanding the impact of design attributes in generating faults. First, the data collection process presented in Section 6.1. Second, descriptive statistics described in Section 6.2 .Then, inferential statistics described in Section 6.3.Finally, the data analysis discussed in Section 6.4.

CHAPTER 7 consists of the thesis conclusion. This chapter starts with summarizing the thesis work and listing our finding. Which is followed by a discussion section that describe the quality of finding, benefit, and suggestions for both Systemite and the data provided companies. Finally, future works are presented.

APPENDIX A includes our first related paper. Using XPath to Define Design Metrics.

PESARO 2016 : The Sixth International Conference on Performance, Safety and Robustness in Complex Systems and Applications. Lisbon, Portugal

APPENDIX B includes our second related paper. Traceability Data in Early Development Phases as an Enabler for Decision Support.

XP 2016 : International Workshop on Emerging Trends in DevOps and Infrastructure. Edinburgh, Scotland.

2
Related Work

In this chapter, different related literature is described. The chapter begins by introducing the method used for selecting relevant literature, then it describes the fault prediction at different stages of development process, which are the implementation and design phases. Thereafter, design metrics and its uses for predicting fault are presented. Furthermore, traceability between system artifices described. Finally, software fault prediction models are presented.

Foreword

Fault prediction is the process of evolving models that can be used by the software practitioners in the early stage of development life cycle for detecting faulty constructs such as, classes, requirement, and modules [37]. The process of software fault prediction can be done in different phase of development process. Accordingly, the software community have been investigated the fault proneness in all phases of system development. However, predicting the fault in the early stage of the development process is always desired due to its benefits in reducing the cost and improving the quality [16]. Moreover, it enhances the reliability, maintainability, efficiency, and portability [29], and of course, reduce the effort of testing, and increase the performance [30].

2.1 Literature Collection

A literature review serves many important purposes, including establishing the knowledge body for the research, framing the valid research methodologies, expanding the knowledge of the researcher, and preventing the researcher from conducting research that already exists [47]. As a matter of fact, a research is a small piece in a complex jigsaw puzzle [46]. In our project the method of conducting the literature review involved the following steps [47]:

- Identify search keywords: it was obtained through the following, a pre-study held at the start of the project, supervisors guidelines, superusers and industrial experts input, and old related project held by the hosting company.
- Literature databases & electronic resources: this included Chalmers databases which include 47 databases in computer science area, and many other related databases, i.e, statistic and research. Industrial literature which was available at Systemite site. Physical resources which were available at both Chalmers library and Systemite.

- Keywords search: The search process had started first by reading title, article's keywords, abstract, and conclusion. Then, for those related work by reading the whole work, even go through their references. In addition, during this work we had two publications, hence we received several recommendations to examine several related works.
- Know the literature: by listing, defining, describing, identifying, and extracting the meaningful information from the selected literature
- Comprehend the literature: by summarizing, differentiating, interpreting, and contrasting the related work.
- Apply the literature: by demonstrating, illustrating, solving, relating, and classifying , as descried on chapter 4 and 5.
- Analyze the literature: by separating, connecting, comparing, selecting, and explaining the related works.
- Synthesize the literature: by combining, integrating, modifying, rearranging, designing, composing, and generalizing the related works.
- Evaluate the literature: by assessing, deciding, recommending, selecting, judging, explaining, discriminating, supporting, and concluding the related works.

2.2 Prediction of Fault at Implementation Phase

There have been huge work exists on using code properties as predictors of faults. Many works, for instance, use software metrics like code complexity, code size, and so on. Some exhaustive literature reviews exist, like for instance, the work of Radjenovic et al. [12]. They aim at identifying software metrics and to assess their applicability in software fault prediction. For that, 106 papers were selected published during two decades (between 1991 and 2011). The selected papers were classified according to metrics and context properties. The results of this review according to the select studies were as the following [12]:

- Object-oriented metrics were used (49%) nearly twice as often as traditional source code metrics (27%) or process metrics (24%).
- The most popular object oriented metrics were the CK (Chidamber and Kemerer's) metrics, which were used in almost (50%).
- There are significant differences between the metrics used in fault prediction performance.
- Object-oriented and process metrics have been reported to be more successful in finding faults compared to traditional size and complexity metrics.
- Process metrics seem to be better at predicting post-release faults compared to any static code metrics.

Another literature Review in this context, "using code properties as predictors of faults", is the work of Catal et al. [10]. In this paper the authors reviewed 90 software fault prediction papers published between year 1990 and year 2009 in software fault prediction for both machine learning based and statistical based approaches. According to this study, they found the following:

- The majority of the studies used method level metrics, i.e., numbers of lines of code.

- Machine learning techniques were the mostly based models comparing to other techniques.
- Naive Bayes is a robust machine learning algorithm for supervised software fault prediction problem.

2.3 Prediction of Fault at Early Phase

Moving to the design level properties (or requirements and architecture level properties) as predictors, which is our interest domain, some work exists but the number is significantly less compared to the work on code properties. One of those work is [15], in this paper the authors present the need of fault prediction at design phase to decrease the cost of development, reduce the effort of testing, and increase the performance. In addition, they stated that fault prediction at early stage could reduce the risk of a new project, where fault could be discovered earlier and knowledge can be transfer between different projects, by using design metrics results of other projects as initial guideline for the new project. In this study they used a data set from NASA MDP repository from seven public domain software development at design level. They conclude based on the data studied “The metrics from the early software life cycle are useful and should be used, regardless of the data from within project, software fault prediction model can be built at design phase with other project’s design level fault data” [15].

Moreover, there are a number of studies on software fault prediction using the requirements level properties. For instance [19], in this paper the authors investigate whether metrics available early in the development lifecycle (requirements phase) can be used to identify fault-prone software modules. Starting from this question and using data from three NASA projects, they found that metrics applied in early stage of project lifecycle can have a important role in project management, either by pointing to the need for increased quality monitoring during the development process or by using them to assign verification and validation activities [19].

Another research carried by Singh et al. [30], in this research the authors evaluated and analyzed techniques to predict the fault at design phase. Vast experiments were performed on eleven projects from NASA metrics data program, which offers design metrics and its associated faults, to discover the effect of different elements of a learning scheme for fault prediction. They found that design metrics can be used precisely as software fault indicator in early stage of software development.

2.3.1 Design Metrics

Design metrics have been have been developed to help in predict software defects or evaluate design quality for a long time [11], [30], [31], [32], and [33]. Those design metrics are described in details in Chapter 4. Some of these metrics show a better ability to discover faults, Rathore et al. [11], they suggested that models built on coupling and complexity metrics are better and more accurate than those built on using the other metrics(cohesion, inheritance, and size). They draw this conclusion based on empirical study. In our work, we extract most of the metrics that we implemented from literature [11], [18], [20], [21], [22], [26], [19], [29], [19] [30], [31], [32], and [33]. as well as from the best practice of the ongoing and completed projects at Systemite [6] and [48].

One of those project is Synligare [56], it is European project cared by Systemite and four other companies (Volvo, Semcon, Arccore, and Autoliv), the goals of the project are complexity bridging, cost effective, and collaborative development. In this project they look attentively at metrics as a key performance indicators used to gauge and follow/up system development status. For them metrics is important to have early feedback from customer to the supplier, and to be able to assess progress for the OEM (original equipment manufacturer). Accordingly, they define large number of metrics and categorise them in three categories, which are [48]:

- Qualities and correctness, i.e., Dependability and Cost.
- Progress and completeness, i.e., Number of completed work products and Number of change requests
- Risk and impact, i.e., Appropriateness of method and Impact of monitored system.

Another related work for this project is [6], in this work the authors proposed a new way to define design metrics using Xpath, to read more about Xpath [50], as well as, they presented six metrics that measure both the progress of requirement allocation, and system complexity. These metrics are :

- The set of all requirements.
- The set of allocated requirements.
- The set of unallocated requirements
- The fraction of the sets.
- Cyclomatic Complexity.
- Couplings between objects.

2.4 Traceability

Traceability can be described as the ability to track and follow processes that links and depends with one another to complete a certain job [57]. A traceable pathway from the root of fault (design properties) to a leaf of test case fail is necessarily to specify the cause behind generating fault. Over the last decade, researchers have studied on particular areas of the traceability problem and they tried to develop more complicated tooling, even they suggested many new research areas to be addressing [34]. Traceability has many uses, it could be used to check if the requirements have been satisfied in advance development phase, like design and code. Also, it could help to assess and manage the impact of changing a system artifact, i.e., requirement, among many other artifacts [35]. Moreover, traceability has been used in fault based testing to generate test data to demonstrate the absence of a set of pre-specified faults [36].

2.5 Software Fault Prediction Model

Software fault prediction models have an important role in software quality assurance. They identify software artifacts, i.e., modules, components, requirement, etc. which have a higher chance of failing a test. These artifacts, in turn, receive additional resources for verification and validation activities [28]. The system design metrics and fault data can be used to build models that can be used to predict faulty modules in the early stage of software development process. System fault prediction model can be done by categorizing the system modules as fault prone and not fault prone. The model's metrics and fault data can be acquired from similar product or other version of the product [37]. Many researcher, for example [37] and [54] have presented the use of various machine learning techniques for the software fault prediction problem. Machine learning techniques have the ability to predict software fault proneness and can be used by software users and researchers. An example for using machine learning techniques to predict software fault is the work of G. John et al. [55]. The employed random forests (RF) method ,which is an extension of decision tree learning, for prediction of faulty modules with data sets provided by NASA .

3
Methodology

In this thesis project we use the quantitative case study conducted together with our industrial partner Systemite AB, and the raw data was provided by two Swedish automotive companies, that use Systemite's platform to store and manage their data. The chapter organized as follows, Section 3.1 presents the overall methodology. Section 3.2 further describes the research methodology steps. Finally, Section 3.3 discusses the research limitation and validity threats.

3.1 Methodology

This section presents an overview for the methodology of our work. The thesis work is conducted using the empirical research method [40] and [41] based on the quantitative approach [42]. First, we studied the raw data, which was provided by two Swedish automotive company and sorted using SystemWeaver [43], with the goal to identify whether the relationships between requirements and design artifacts, as well as, information from individual artifacts can be leveraged to predict software faults. Our hypotheses was based on the assumption that system design attributes affect the probability of failing test case. More precisely, we have six hypotheses cover system, component, and requirement design attributes, as mentioned in Section 1.2.

The next step after defining the research goal and hypotheses, we conducted a thorough case study analysis [52]and tested the suitability of the selected metrics, which presented in Chapter 4. For that purpose, we used both the available body of knowledge existing in the literature and the expert people at the hosting company (Systemite), as well as, the expert users at the customer side (Systemite's customers).

After defining the bases of our project, the next step was to understand the raw data and filter them in order to apply our measurement. This step was a challenge for us, where the size of the raw data was too large and it comes from different sources. Even, all data providers use the same platform, but every one of them customized the platform to fit their own needs, by defining their own meta-model. To overcome this challenge, we used Xpath expression[6], and this was the first part of our implementation, as shown in section 5.2.

Then, in order to perform the intended measurements and present the results, a tool has been implemented, we called it SmartTrace, as shown in section 5.3. which is able to apply the system, component, requirement, testing, and combined metrics, using the available traceability data, as described in Chapter 4. As a part of metrics and tool validation, we conducted several interviews with both SystemWeaver's developers and users. A detailed description of the research methodology steps delivers in the next section.

3.2 Research Methodology Steps

This section presents a detailed description of our research methodology steps. The research methodology includes ten steps, in general the result of each step was the input for the next one. But, this was not the case all the times, in some cases we modified a step based on a feedback or a result of other steps, as well as, some times more than one step could be a prerequisite to start other step, as shown in Figure 3.1.

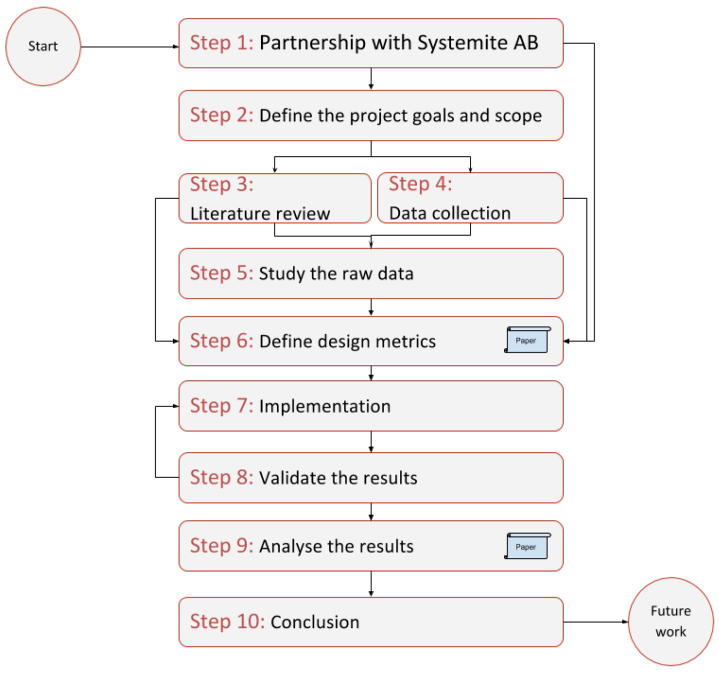

Start
Step 1: Partnership with Systemite AB
Step 2: Define the project goals and scope
Step 3: Literature review
Step 4: Data collection
Step 5: Study the raw data
Step 6: Define design metrics
Paper
Step 7: Implementation
Step 8: Validate the results
Step 9: Analyse the results
Paper
Step 10: Conclusion
Future work

Figure 3.1: Research methodology steps.

Figure 3.1 shows our research methodology steps, and the following are the description of those steps:

Step 1: Partnership with Systemite AB [44], at the start of this thesis we built a partnership with Systemite. The selection of this hosting company was due to several reasons. These reasons are Systemite's powerful platform [43] (SystemWeaver), the availability of the raw data for more than 15 years, Systemite's previous research and project, and common interests between our thesis gaols and their work.

Step 2: Define the project goals and scope, after building the partnership we define the thesis goal and scope, based on the common interests between us and the hosting company, properties of the available data, Systemite's previous projects i.e., Synligare [48], and our previous industrial project with Systemite, where we used this industrial project as a foundation stone to start this thesis.

Step 3: Literature review, at this step we build a body of knowledge, position our work, and presents the need for this thesis, as descried in Chapter 2.

Step 4: Data collection, this step was going in parallel with literature review step, because we wanted to get a feeling about the available data, to pick from literature what it is relevant, as described in Section 6.1.

Step 5: Study the raw data, this step was one of the most challenging step for many reasons, first the size of the data was too large. Second, understanding the platform where the data is stored, since SystemWeaver has 18 modules supporting the system development process, and gaining a good understanding about them took long time. Third, data inconsistency, because the data comes from different sources. Fourth, no single source has an overall understanding about the data, because the system built by different people located in different site, even from different companies, i.e., meta-model built by the application engineers, testing by system tester, and system architecture by system designer. To comeover those challenges we interviewed several people from both the platform development company and their customers, as well as, we used Xpath expressions [6], (Xpath [50]), to study the data and have a better overview, as described in Section 5.2.

Step 6: Define design metrics, at this step we select 19 metrics to be implemented in our work, as described in Chapter 4. The selection of these metrics was based on three criteria, first, Systemite's previous work and research, including our previous industrial project with them (step 1). Second, studying the related work (step 3). Finally, the properties of the studied data (step 4). At the end of this step we published our first paper [6], *"Using XPath to Define Design Metrics"*.

Step 7: Implementation, this step includes the tool implementation, which has been used to extract the data and build our data set, we called it SmartTrace, as described in Section 5.3.

Step 8: Validate the results, the gaol of this step was to make sure that the result of the metrics are correct and accurate, as described in Section 5.3.1.

Step 9: Analyse the results. After validate the result we applied both the descriptive and inferential statistics on the data set that we get, as described in Chapter 6. At the end of this step we published our second paper [39], *"Traceability Data in EarlyDevelopment Phases as an Enabler for Decision Support"*.

Step 10: Conclusion, finally we draw our conclusion based on the statistical result of the industrial case study, as described in Chapter 7. At the end of this step we presented our future work and suggestions for both Systemite and the data provided companies.

3.3 Limitation & Validity Threats

The validity of a study indicates the trustworthiness of the study results, and to which degree the results are not biased by the researchers' subjective point of view [52]. This section describes the threats to validity in this thesis based on the classification scheme provided by Runeson et al. [52], which includes internal validity, external validity, conclusion validity, and construct validity.

3.3.1 Internal Validity

Internal validity is a scientific concept that address the relationship between two variables [52]. In other words, it is the extent to which a researcher proves that only the independent variable is responsible for change in the dependent variable . In our project the independent variables are the result of metrics which measure the system design attributes, i.e., number of component in port or size of the system, and the dependent variable is the percentage of failed test cases, i.e., a component failed 20 % of the total performer test cases. Our goal is to investigate the relationships between requirements and design artifacts, as well as information from individual artifacts(independent variable) and the test case result(dependent variable).

This project has two threats to internal validity, which are selection and maturation. Starting with sample selection, the studied data comes from two automotive companies that provided us with three mature systems, after the data filtration process we end up with a data provided by one company, this data is 12 versions of a same system, as described in Section 6.1. Having a data from one company, as well as the data is versions of the same system could effect the result of the study. Moving to the second internal threats which is maturation, as mentioned before the data comes from system versions which mean that the system maturation could effect the number of failed test cases, where system engineer become more experience and mature over the system development period, especially that the period between those 12 versions is about five years. On the other hand, this long period could reduce the threats of other internal validity, such as history, because of the isolation of effect an interceptor happened, i.e., crisis or company's unstable situation.

3.3.2 External Validity

External validity is the extent to which the results of a study can be generalized to the world at large [52]. In our work the data comes from automotive industry, and our finding could not be generalized to other industries. Moreover, the data stored at SystemWeaver, which mean that the internal design of SystemWeaver could effect the study result. As a part of future work we had a plan to study more data provides by other companies, as well as other industries, in order to generalize our finding. At the present time we repeat our measurements over 12 versions of a system, and the tool that we developed is capable to measure any system, as long as the system is build using a same meta model, even if a system developed based on different meta model the tool could be modify easily to catch the new change. At the same time the tool has the capability for expand to measure more system design attributes, because we implement every primarily blocks, to reuse them later when we need to define new metrics.

3.3.3 Conclusion Validity

Conclusion validity is the degree to which conclusions we reach about relationships in our data are reasonable [52]. More precisely statistical conclusion validity refers to whether or not we have used our statistical properly and have drawn the right conclusion based on our results. In our work not all studied versions show the correlation between the dependent and independent variables. As a matter of fact we have two conclusion threats, which are restriction of range and random selection. First, restriction of range, in data collection step we excluded all system artifacts that were not tested or did not have a traceability link to other system artifacts, as described in Section 6.1. This could restrict the range of data selection, as well as the select data was built based on a meta model where a system could be define only according to this meta model, which mean that the meta model could restrict our data. Second, random selection, the selection of our data had four limitations. These limitations are the data come from one company, the data come from one industry, the data are versions of same system, and the data stored in SystemWeaver according to specific meta model.

On the other hand, we overcome other statistical conclusion threats, such as violated assumption of statistical test and unreliability of measures. Starting with statistical assumption, we make sure that all our assumption about the data are correct, by applying several statistical test, i.e., normality test, linearity test, etc. as discussed in Sections 6.3 & 6.2. Moving to, unreliability of measures, we ensure that our measurement is accurate and correct using three practises. First, review the related work to make sure that the applied metrics measure what it is intended to measure. second, interviewing both Systemite's experts (application engineers from the platform development site) and SystemWeaver's users (system developers from the automotive company where our data comes from), in these interviews we presented our work, described our assumption, and collecting feedback. Finally, tool validation by comparing the results of our work with other tool that offer similar functionality, for example SystemWeaver's script language to validate the metrics results and SPSS to validate the statistical result.

3.3.4 Construct Validity

Construct validity means how well a test or tool measures the constructs that it was designed to measure [52]. As mentioned before we make sure that our metrics measure exactly what they intend to measure, to do that we used both human expertise and tool support as described in Section 5.3.1.

4
Metrics

This chapter presents the design metrics. These metrics were implemented in our work, in order to investigate the relation between design attributes and its relation to generate fault. The chapter organized as follows, Section 4.1 presents system metrics. Section 4.2 presents component metrics. Section 4.3 presents requirement metrics. Section 4.4 presents testing metric. Finally, Section 4.5 presents combined metrics.

Foreword

In our work, the metrics selection method is based on three parts. The first one is to study earlier academic work [11], [18], [20], [21], [22], [26], [19], [29], [19] [30], [31], [32], and [33]. The second is to get best practice from ongoing and completed projects at Systemite [6] and [48]. The third is the attribute of the studied data [43], as descried in Section 5.1.1. Provided that, we select 19 metrics, which divided later into four main categories, in addition to fifth one where we combined metrics from the previous four main categories. Those metrics categories are system, component, requirement, testing, and combination categories.

4.1 System Metrics

In this section, we presents three system metrics that we implemented to measure the complexity, weight, and size of the system. These metrics are cyclomatic complexity, weight(number of function, requirement, and edge), and size(number of components).

4.1.1 Cyclomatic Complexity

Cyclomatic complexity(CC) is a software metric, used to indicate the complexity of a program. It counts the number of independent path through a system [23]. The higher number mean more complex system, also it could be used to give an indicator about the number of test cases needed to cover the system.
In other words, the relation between system modules, represents the design structure also define the overall design complexity. McCabe et al. [22] concluded that, quantifying the complexity of a system design provide the designer a metric, could forming a significant management tool. Many following studies, support this claim

and provide statistical evidence for that. For instance Jiang et al. [19] and Rathore et al. [11]. The formula of calculating cyclomatic complexity is [23]:

$$CC = E - N + 2P \tag{4.1}$$

Where:

- E = the number of edges of the graph.
- N = the number of nodes of the graph.
- P = the number of connected components.

4.1.2 Weight

A research made by Chidamber et al. [20] addresses the need for design metrics, by proposing a suit of metrics contended six metrics one of them was Weighted Methods Per Class (WMC), which is the number of method implemented by a class. Their intention to define this metric is to address two points. The first one is number of method in a class could give an indicator of resources needed to develop and maintain. The second one, large number of method in a class will affect its child, since child will inherit those methods. Also, the large size will affect the possibility of reuse, since it is more application specific [20]. We implemented three weight metrics measure number of system function (methods), number of requirements, and number of edges, as described in Table 4.1.

System Metrics	Description
Number of functions	The number of functions per system or sub-system. In our work, *Function* : is an artifact that specifies and represents an end user task.
Number of requirements	The total number of requirements per a system. In our work, *Requirement* : is an artifact that specifies behavior or function of a system.
Number of edges	The total number of edges per system. In our work, *Edge* : is an artifact that connect two components together, by connecting a component out port with other component in port.

Table 4.1: System weight metrics.

4.1.3 Size

As in source code metric we have a number of line code (LOC) as a measurement of system size, number of component per system is a similar metrics but on design level. It gives an indicator about the size of a system, whereas large size means more complex and higher probability to find error comparing to small size [18].

4.2 Component Metrics

There are several types of component metrics, which measures the component from different perspective. For example, the number of component incoming ports, or the component interaction. In this section we present two types of component metrics, which we implemented in our work, which are interaction (component interaction, actual interactions, and total interactions performed), and weight (number of in port, out port, total port, requirement, responsibility, and function).

4.2.1 Interaction

Interaction metrics are aimed to measure the communication between a component and other surrounding components. Chen et al. developed many different types of component interaction metrics [18]. We implemented three of them in our work, which are component interaction, actual interactions, and interactions performed metric.

4.2.1.1 Component Interaction

Component interaction metric(CI) measures a component interaction in individual level. It is defined to compute the ratio of dividing the number of component incoming ports by the number of outgoing ports. The formula is [18]:

$$CI = \frac{I}{O} \tag{4.2}$$

Where:

- I = The number of a component's incoming ports.
- O = The number of a component's outgoing ports.

4.2.1.2 Actual Interactions

Actual interactions metrics (AI) measures the interaction intensity among the whole components. By measuring the actual component interaction comparing to the maximum interaction among the whole system components. The formula is [18]:

$$AI = \frac{I + O}{I.Max + O.Max} \tag{4.3}$$

Where:

- I = The number of a component's incoming ports.
- O = The number of a component's outgoing ports.
- I.Max = The maximum number of a component's incoming ports over a whole system.
- O.Max = The maximum number of a component's outgoing ports over a whole system.

4.2.1.3 Total Interactions Performed

Total interactions performed metric (TIP) measures the ratio of actual interaction divided by the total number of components in a system. The formula is [18]:

$$TIP = \frac{AI}{C} \tag{4.4}$$

Where:

- AI = The actual interactions (Equation 4.3)
- C = The total number of system components.

4.2.2 Weight

In like manner to what we did in system weight metrics (section 4.1.2)but this time in component level, we implement six component weight metrics, which presents in Table 4.2

Metrics	**Description**
Number of in port	The number of incoming ports per component. In our work, *In port* : is an interface that allows component to receive information, i.e., signal, from other component.
Number of out port	The number of outgoing ports per component. In our work, *Out port* : is an interface that allows component to send information, i.e., signal, to other component.
Number of ports	The total number of ports per component. In our work, *Total ports* = incoming ports + outgoing ports.
Number of requirement	The number of requirement per component. In our work, *Requirement* : is an artifact that specifies behavior or function of a component.
Number of responsibility	The number of responsibilities per component. In our work, *Responsibility* : is an artifact that is responsible for one part of the component's job.
Number of function	The number or functions per component. In our work, *Function* : is an artifact that specifies and represents an end usertask.

Table 4.2: Component weight metrics.

4.3 Requirements Metrics

Requirement management is an essential part of system development process. An empirical study tested many projects from different organization, showed that more than 50% of defects generated from requirement phase [29]. Another research claimed that measurement is essential to ensure the quality of requirements to support the rational management decisions [26]. The software community has developed a large number of requirements metrics to ensure high quality requirements and measure the requirements attributes.

Jiang in his research investigated how to use requirement metrics in early stage of development process to insure the quality of the system and predict fault. He used several requirement metrics, and concludes that using textual description of requirement alone could not be a good enough indicator to detect defects, comparing with use those metrics together with other design metrics [19].

In our work, we implemented two requirement metrics, both of them are quantitative based metrics, taking the advantages of the availability of traceability data. We left the requirement textual interpretation metrics for future work, based on the available resources for this thesis project. The implemented metrics are number of sub-requirements per requirement and number of linked components metrics.

4.3.1 Sub-Requirements per Requirement

It measures the number of sub requirements per a requirement. This metric could have other uses beside fault prediction, such as defining the weight of the parent requirement, assessing the re-usability, and evaluating the changeability.

4.3.2 Number of Linked Components

It measures the number of different components that use the same requirement. For instance, this metric will be equal to one if the corresponding requirement is used by one component, and five if it is used by five different components.

4.4 Testing Metrics

Testing is one of the most important procedures in quality assurance process. However, it is a costly process and it requires special skills. Moreover, some defects are very hard to discover, and with time they become very costly to fix [16]. In our work we implemented three testing metrics, which are total number of test cases, number of passed test cases, and number of failed test cases. In the light of these three metrics we can extract more metrics, i.e., percentage of failed test case by dividing the number of failed test case by the total number of performed test case for a specific system artifact.

4.4.1 Number of Test Case

It measures the total number of performed test cases on a system artifact, i.e, component or requirement, or on a whole system, regardless the result of the test case.

4.4.2 Number of Passed Test Case

It measures the number of passed test case, which performed on a system artifact or a whole system.

4.4.3 Number of Failed Test Case

It measures the number of failed test case, which performed on a system artifact or a whole system.

4.5 Combination Metrics

Every single metric has its own value, however combining them together creates a benefit that cannot be obtained by any individual one. Many researchers investigated the synergy between metrics, for instance, Hristov et al. [21] developed a re-usability model, by forming a single metric measures several factors influencing the component re-usability, which are reuse, adaptability, price, maintainability, quality, complexity, documentation, and availability.

Another example is to use complexity and testing metrics together to measure how well a system is tested. Given that cyclomatic complexity metric may give an indicator about the number of test cases needed to cover the system [23], along with testing metrics like total number of test case performed, a system stakeholders could have a better understanding about the system quality and testing progress.

5 Implementation

This chapter describes the software that was implemented. The development work could be divided into two stages. First, preparation of the study and data cleaning using Xpath expression, as shown in Section 5.2. Second, implementation of the study using C# WPF, as shown in Section 5.3. The chapter begin with describing the hosting company and their platform. Then, it describes the use of Xpath expression to define design metrics and filtering our raw data [6]. Finally, it presents the tool implementation, which has been used to extract the data and build our data set, we called it SmartTrace [39].

Foreword

All data used in this project had been provided by two Swedish automotive companies and it is based on SystemWeaver platform deployed to different types of vehicle. In order to perform the measurements and present their results, a tool has been implemented which is able to apply the design metrics, described in Chapter 4, analyse the results of the metrics by applying statistical test, and visualized their result. The ultimate goal is to build a decision support system based on historical product data, which guide the system developer during the design process [39].

5.1 Systemite AB

In this section, we introduce our partner Systemite AB [44], and their platform SystemWeaver [43]. SystemWeaver is the systems engineering platform that we use to conduct this thesis project, where all the raw data were stored in this platform.

SystemWeaver is an information management platform solution for systems engineering and software development [43]. SystemWeaver is developed by Systemite AB [44], a Swedish software development company located in Gothenburg with one branch on Stockholm and two representations, in Republic of Korea and China. SystemWeaver allows user to design system in different abstraction level and keep traceability between all system artifacts, create product specifications, managing requirement, generate reports, managing version and configuration, and testing solution. In addition, it supports work with product lines including feature models, defining various models and their variability. SystemWeaver has been used by different companies for approximately 15 years, which create a huge data repository. The goal of this master thesis is to use this data, as well as the platform's powerful to investigate the relation between system design artifacts, as well as information from individual artifacts, and software faults, using the traceability link between requirement, design, and test.

5.1.1 SystemWeaver's Architecture

SystemWeaver is a model driven development environment where data is ruled by a strong meta model that specifies how models could be built. The strong meta models in SystemWeaver means that user could not do anything that is not explicitly allowed by the meta model [43]. Figure 5.1 shows the meta model for SystemWeaver's conceptual architecture, which includes the following [49]:

- Object: it represents all references based on using IDs, i.e., part reference type.
- Generic node: it is a built-in support for structured model(classical product data management), i.e., test specification result.
- Item: it is a basic reusable artifact in SystemWeaver, i.e, function, use case, signal, and requirement.
- Part: it is a relation from one item to another, i.e., send poert, receive port, and sub-function.
- Attribute : it is a typed values that are unique to an item, i.e., ID, name, and status.

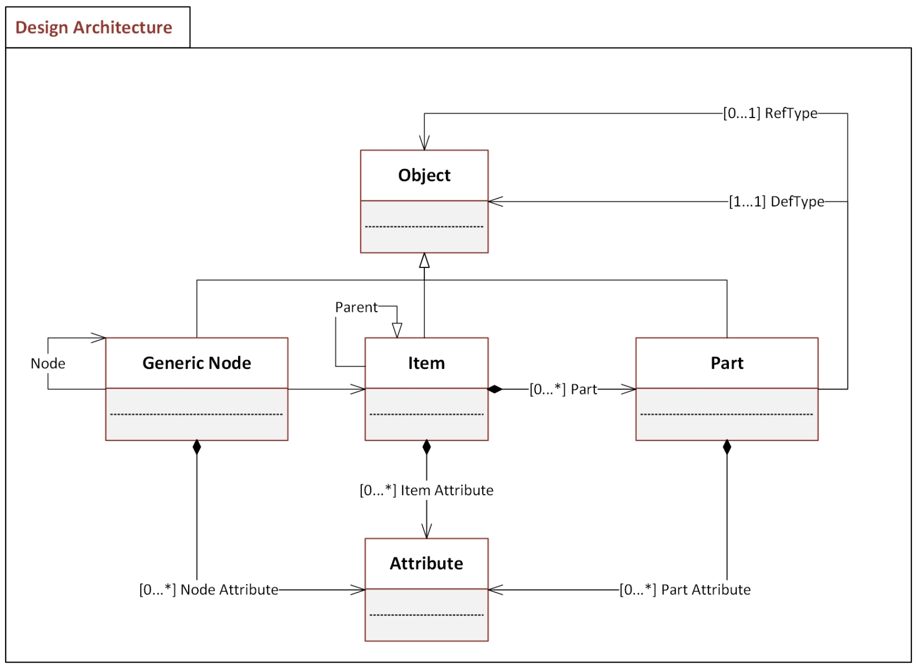


Figure 5.1: SystemWeaver's conceptual architecture meta model.
Source: Systemite AB, SystemWeaver's help documentation.

Similarly, Figure 5.2 shows the meta model for SystemWeaver's testing and verification, which includes the fowling [49]:

- Test suite: it is the higher abstraction level of testing, which includes a collection of test cases intended to test a behavior or a set of behaviors of an abstract artifact.

- Test: it is a part of test suite point to a system under test.
- Test scope: it describes specifically what the test intended to accomplish.
- Test specification: it includes both test cases and test specification requirement.
- Test case: it is an artifact intended to test part of system behaviors.
- Abstract artifact: it is a system under test.
- Requirement: it is a test specification requirement, which the test case intented to test.

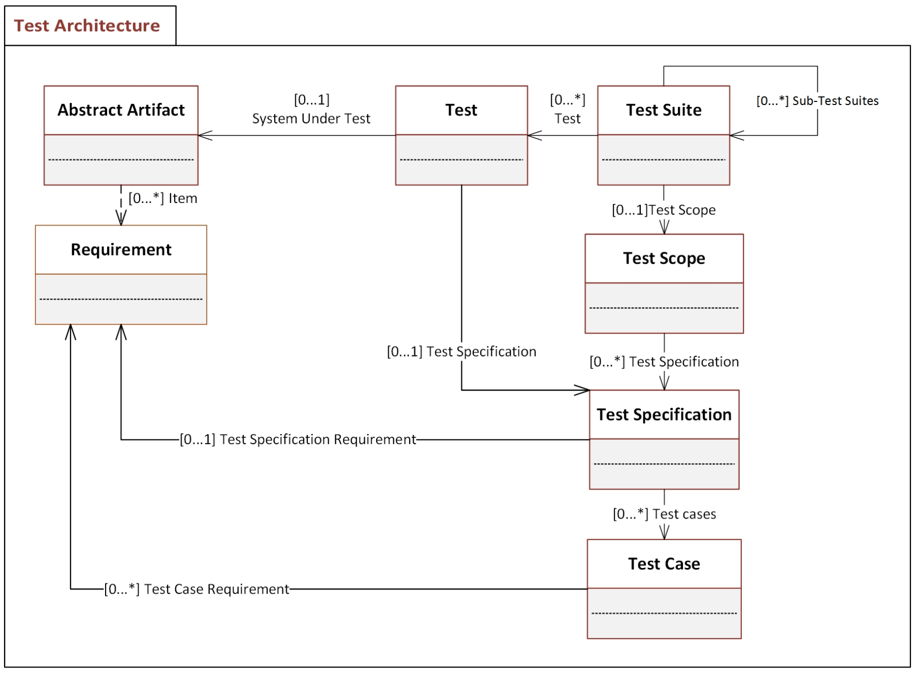


Figure 5.2: SystemWeaver's testing and verification meta model.
Source: Systemite AB, SystemWeaver's help documentation.

Moving to traceability in SystemWeaver, the traceability data takes different forms, as shown in Figure 5.3 [39]. First, it includes the relations (called T.W in the figure) between the product artifacts at the same abstraction level, for instance, the traceability links between a software component, its requirements, and test case validating the component's behavior. Second, SystemWeaver also maintains the relations (T.L in the figure) between the product artifacts on different abstraction levels, i.e., the traceability links between analysis level and logical design level. Finally, the traceability information could include the relations (T.V in the figure) between product versions, i.e., the traceability links between Version 1 and version 2.

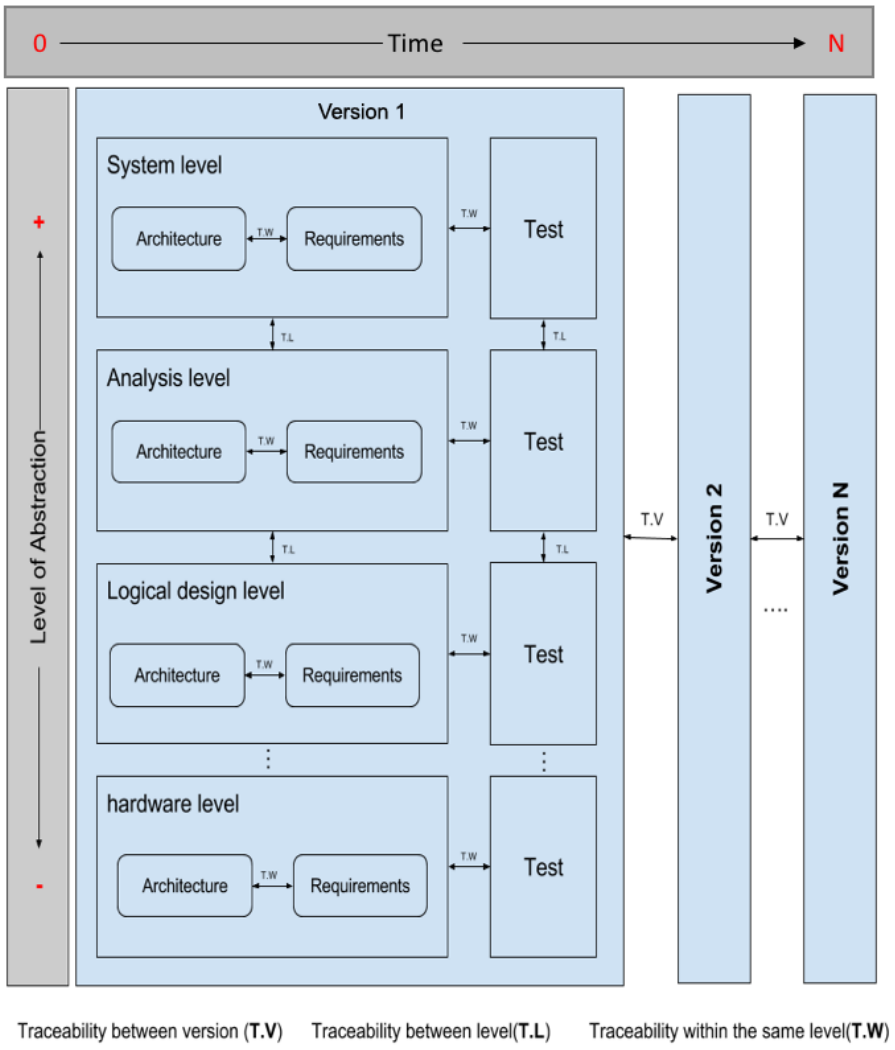


Figure 5.3: SystemWeaver's traceability between system artifacts, abstraction level, and versioning

5.1.2 SystemWeaver's Models

SystemWeaver has many models supporting the development management process, as present in Figure 5.4 . These modules applied on different level of production process. For instance, feature model describes the features that a product could support. While design model contains the superset of all available components that could be included in a product with possibility to automatically generate the specification of a product, by identifying the way that a component relates to features. Another model is requirement management model, which manages all activities related to product requirement, for example versions, configuration, attributes etc. Also, the platform support both failure mode and effects analysis (FMEA) and fault tree analysis(FTA). Moving to hardware management, the hardware representation model containing a superset of hardware component (ECUs), where network design model manage the communication between ECUs through signals. Moreover, the platform offers a collaboration with other environment, For example automotive open system architecture(AUTOSAR) model support mapping a component's interfaces to defined data types for this Autosar system, requirements Interchange Format (ReqIF) model allowing to export any SystemWeaver model to the ReqIF format and DBC model allowing to store all information that describes the network .

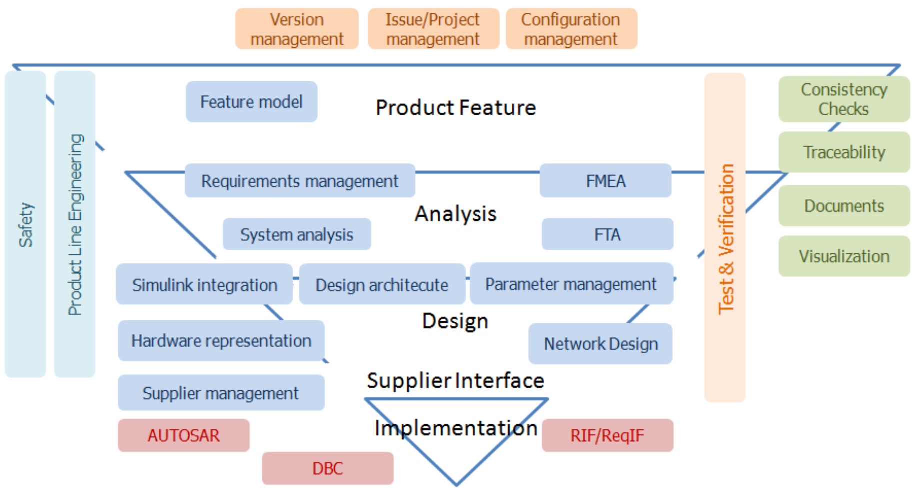


Figure 5.4: SystemWeaver's integrated embedded systems development process. Source: Systemite AB, SystemWeaver's help documentation.

5.2 Xpath Expression

XML Path Language (XPath) 2.0 became a W3C recommendation 2010 [50]. XPath is a specialized query language that can express selection criteria of nodes of an XML document, typically from within an XML style sheet. The selection criteria include the path to traverse in the structure of the document, and additional tests and predicates that must be fulfilled for the selected nodes [6]. The way XPath is used is by :

- Selecting the sets of nodes in the XML document that are relevant for the specific metrics.
- Performing arithmetic operations on the quantities defined by the sets.

SystemWeaver uses language is an XML-based language created to be used by built-in components such as a report generator and a graph generator to extract information from the system. For example, user with architectural privilege could built a report to serve specific purposes, for instance product specification report, this report will be used later by other user to generate a product specification at any point of the development process, even if some requirements or design change, the generated report automatically will includes all those changes.

At the first stage of our implementation we used this script language to get better understanding about the studied data, cleaning the data, define design metrics, and visualize the result. Figure 5.5 shows an example for implementing four metrics using XML path language in SystemWeaver, and Figure 5.6 shows how the result of one of these metrics is presented in a grid view format. The result of those metrics could be presented in different format, such as report and graph.

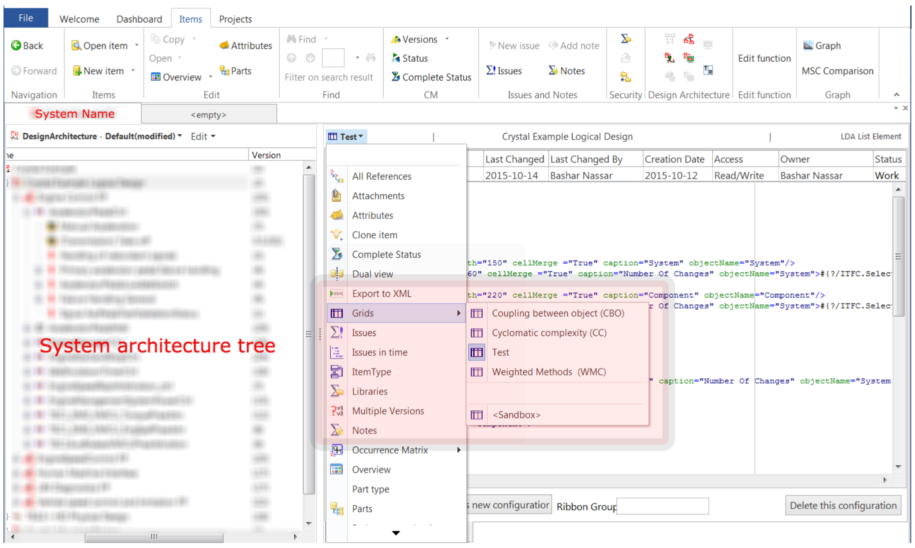
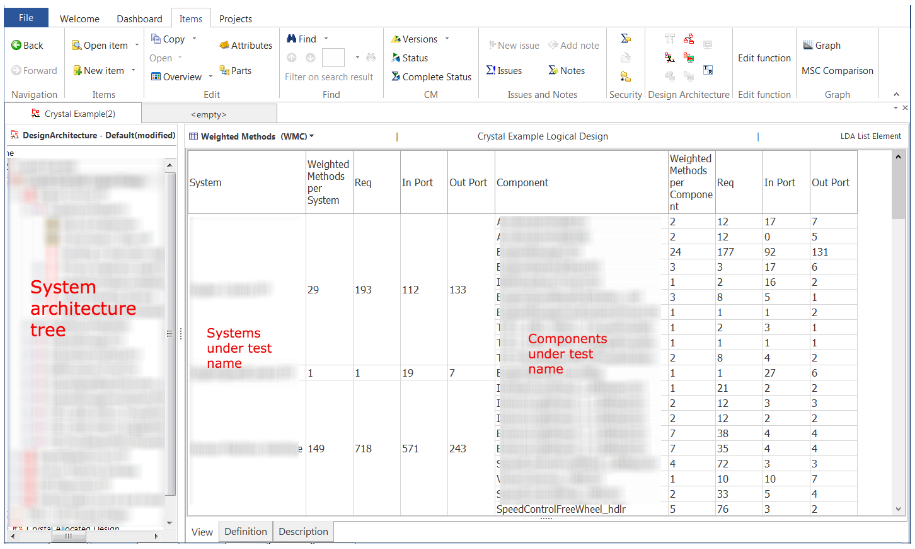


Figure 5.5: Metrics implemented in SystemWeaver.



System	Weighted Methods per System	Req	In Port	Out Port	Component	Weighted Methods per Component	Req	In Port	Out Port
[redacted]	29	193	112	133	[redacted]	2	12	17	7
					[redacted]	2	12	0	5
					[redacted]	24	177	92	131
					[redacted]	3	3	17	6
					[redacted]	1	2	16	2
					[redacted]	3	8	5	1
					[redacted]	1	1	1	2
					[redacted]	1	2	3	1
					[redacted]	1	1	1	1
					[redacted]	2	8	4	2
[redacted]	1	1	19	7	[redacted]	1	1	27	6
[redacted]	149	718	571	243	[redacted]	1	21	2	2
					[redacted]	2	12	3	3
					[redacted]	2	12	2	2
					[redacted]	7	38	4	4
					[redacted]	7	35	4	4
					[redacted]	4	72	3	3
					[redacted]	1	10	10	7
					[redacted]	2	33	5	4
					SpeedControlFreeWheel_hdlr	5	76	3	2

Figure 5.6: Grid view for weight metric in SystemWeaver.

Using Xpath to define design metrics has several advantages as we reported in [6], those advantages are:

- The XPath expressions can be expressed according to the meta-model of the used architecture language, meaning that the correctness of the expressions can be validated statically.
- The XPath language is standardized, technology independent, mature and wide spread.
- A tool implementation of the method may directly interpret and execute the XPath expressions. This makes it easy to try different metrics expressions in the tool implementation, without changing the tool itself.

In addition to these advantages, the implementation of the support for XPath has taken benefit from the fact that XPath supports the selection of sets of elements, thus making it suitable for interactive analysis and traceability between the visualization of the metrics and the underlying data.

On the other hand, there is some natural limitations and disadvantages of using XPath, as we report in [6], and this was the motivation for us to move to the second implementation stage, as described in section 5.3. The limitations of using XPath are the following[6]:

- The approach is likely feasible only for those cases where the language is expressed as XML, specifically, that the schema is a reflection of the used meta-model.
- Given the declarative characteristics of the language it is likely that not all types of metrics can be defined easily in the language.

5.3 SmartTrace

The second stage of our implementation after using Xpath was to build a tool, we called it *SmartTrace*, in order to perform the measurements, present their results in interactive way,visualized the result, analyse the result internally by allowing the tool to implementing the basic statistical methods, and overcome the limitation of using Xpath. This tool ables to apply the design metrics described in Chapter 4, as well as, it is capable to implements a wide range of quantitative metrics, because we developed the basic functions that allows the tool to expand. The ultimate goal is to build a decision support system based on historical product data, which guide the system developer during the design process [39]. In this master thesis we build the basic blocks for this model by develop this tool as a seed for the model. At the same time, the tool was used to investigate our research question and validate our hypothesis.

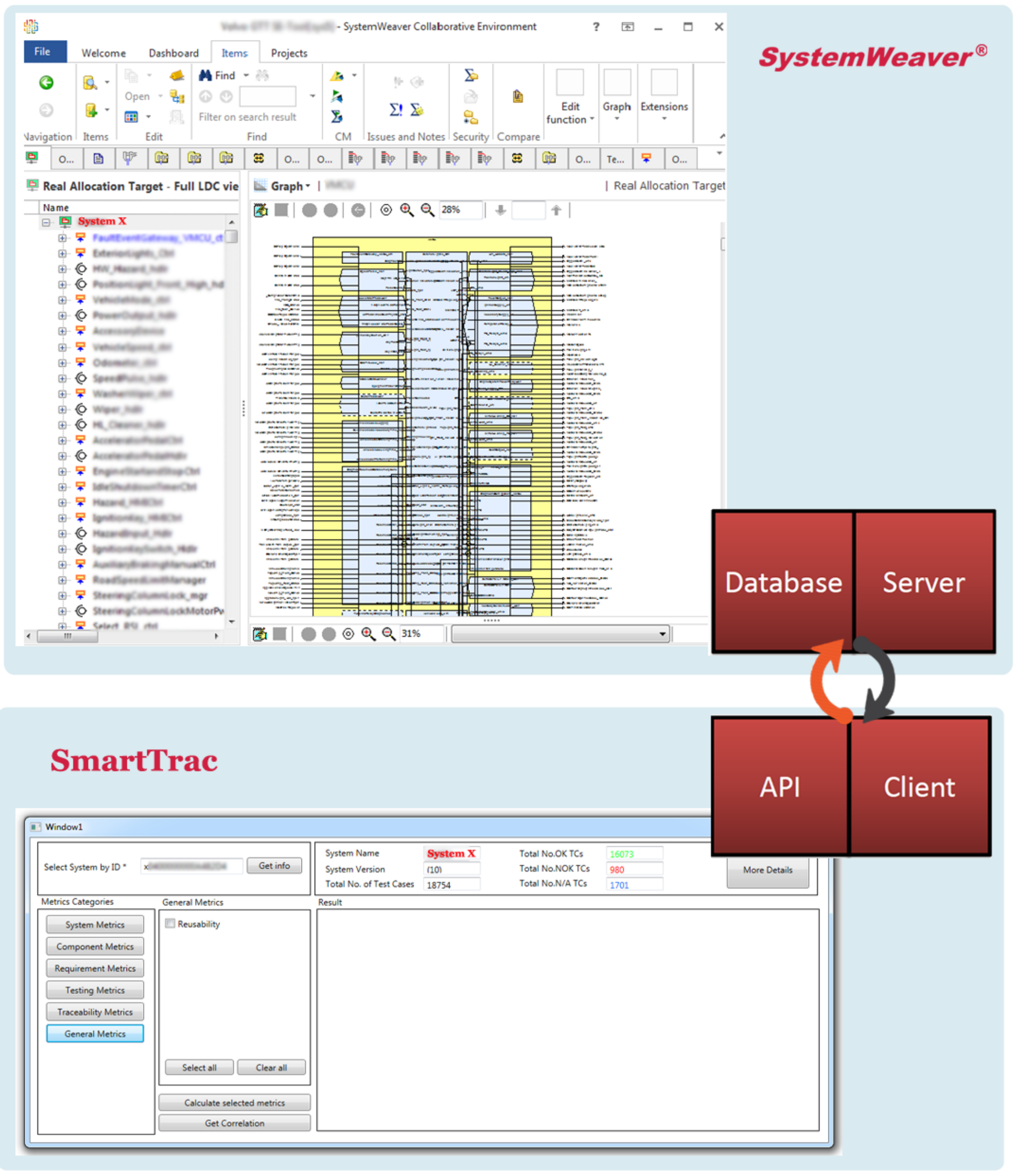
SmartTrace uses SystemWeaer as a data model to retrieve the raw data, where the studied data is stored in SystemWeaver as mentioned before. The connection between SmartTrace and SystemWeaer was build using SystemWeaver's API (application program interface), as shown in Figure 5.7. The API specifies how SystemWeaer's components should interact between each other, as well as how they interact with other external software components. To run SystemWeaver two things are required, first SystemWeaver's server, second SystemWeaver's client. Starting with the SystemWeaver's server part, many pre-requirement should be meet to run the server, which are [49]:

- swTestServer.exe, it is a server application developed by Systemite, that allows user to configure a connection, by choose port, IP, database file repository and SSL (Secure Sockets Layer).
- A database, i.e., SQL Server
- ssleay32.dll, it is a library developed by The OpenSSL and part of The OpenSSL Toolkit.

- libeay32.dll, it is a library that contains encryption functions which allow for coded communications over networks.

Moving to SystemWeaver's client part, several requirement should be exist as well, such as [49]:

- .NET Framework 4
- SystemWeaverClientAPI.dll, it is a library developed by Systemite and offer the basic function and methods, i.e., connection with the server, get item, and get attributes of an item.

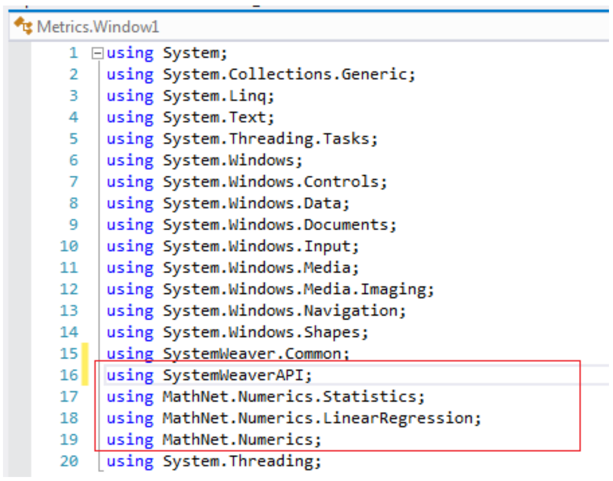


System Name: System X
System Version: (10)
Total No. of Test Cases: 18754
Total No.OK TCs: 16073
Total No.NOK TCs: 980
Total No.N/A TCs: 1701

Figure 5.7: Interaction between SystemWeaver and SmartTrace.

For the sake of statistic we used *MathNet* [51] library, which offers a wide variety of statistical methods, i.e., linear regression and correlation as shown in Figure 5.8. MathNet library allows us to implemented a basic statistical test inside our tool, without the need to export the data to other statistical packages, i.e., SPSS

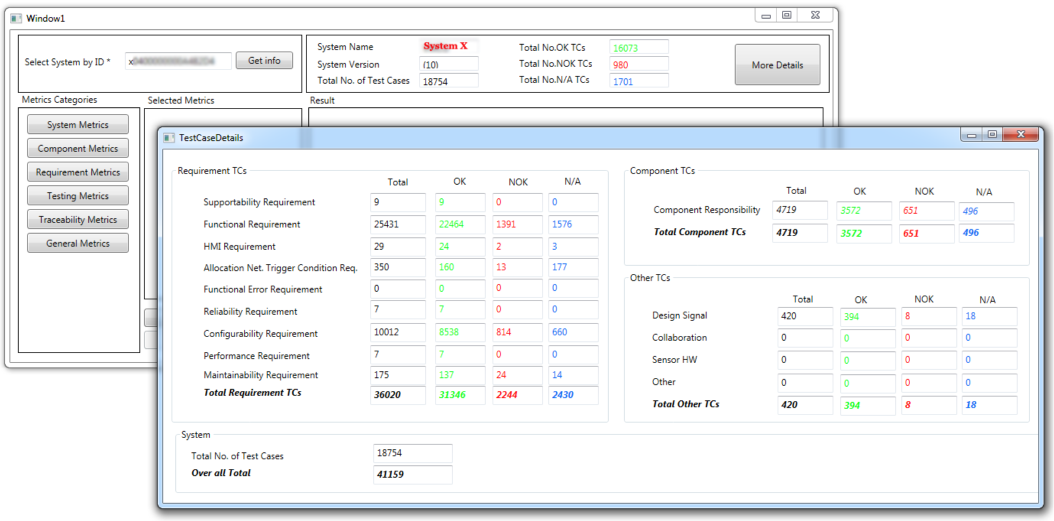
(Statistical Package for the Social Science). Provided that our model become more capable stand alone tool, keeping in mind our ultimate goal to build a full decision support system we implement several methods that we did not used in this thesis. However, for this thesis we Kept using SPSS, because it performs a highly complex data manipulation and analysis, as well as to test our tool, by comparing the result of the tool and SPSS to validate the tool.


Metrics.Window1

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using SystemWeaver.Common;
using SystemWeaverAPI;
using MathNet.Numerics.Statistics;
using MathNet.Numerics.LinearRegression;
using MathNet.Numerics;
using System.Threading;
```

Figure 5.8: SystemWeaver and MathNet APIs.

To give a feeling about the tool and how it works the following figures (Figure 5.9, Figure 5.10, and Figure 5.11) included. They show an example for some of the tool functionalists, Figure 5.9 shows test cases results statistics. It presents how many test cases applied on a system, with further classification based on the test case result, for instance, how many test cases passed, failed, or not complete. Moreover, the tool offer further classification based on the type of system artifact. For example how many test cases applied on requirement, component, etc. This form could be used by system tester to see which system artifact could generate a higher number of fault to use this information in testing prioritization process, as well as it could be used to evaluate the system quality, contracting, progress measurement,.. etc.



Window1

Select System by ID * x[redacted]

System Name	System X	Total No.OK TCs	16073
System Version	(10)	Total No.NOK TCs	980
Total No. of Test Cases	18754	Total No.N/A TCs	1701

Metrics Categories: System Metrics, Component Metrics, Requirement Metrics, Testing Metrics, Traceability Metrics, General Metrics

Selected Metrics

Result

TestCaseDetails

Requirement TCs

	Total	OK	NOK	N/A
Supportability Requirement	9	9	0	0
Functional Requirement	25431	22464	1391	1576
HMI Requirement	29	24	2	3
Allocation Net. Trigger Condition Req.	350	160	13	177
Functional Error Requirement	0	0	0	0
Reliability Requirement	7	7	0	0
Configurability Requirement	10012	8538	814	660
Performance Requirement	7	7	0	0
Maintainability Requirement	175	137	24	14
Total Requirement TCs	***36020***	***31346***	***2244***	***2430***

Component TCs

	Total	OK	NOK	N/A
Component Responsibility	*4719*	*3572*	*651*	*496*
Total Component TCs	***4719***	***3572***	***651***	***496***

Other TCs

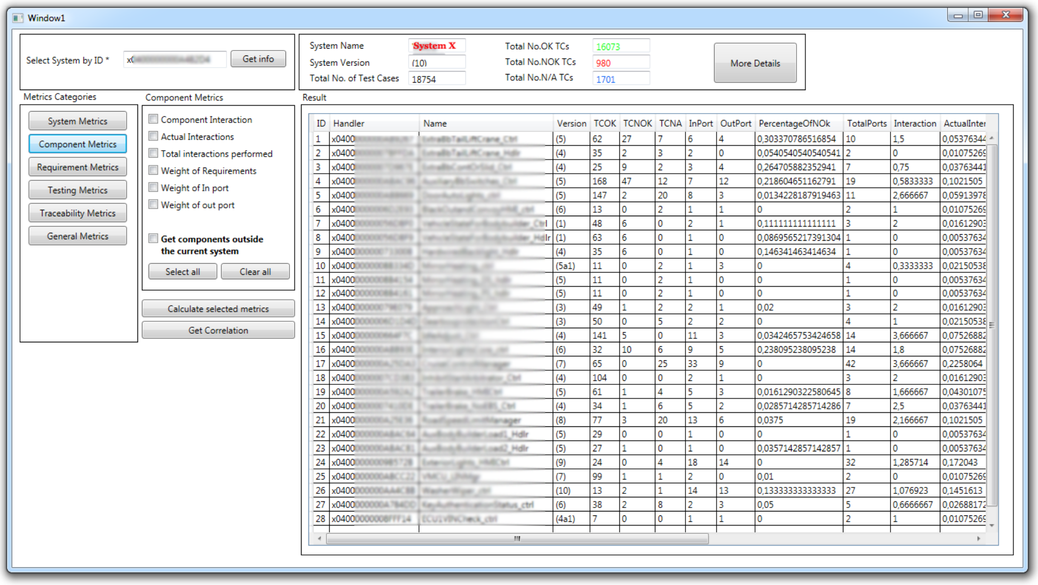
	Total	OK	NOK	N/A
Design Signal	420	394	8	18
Collaboration	0	0	0	0
Sensor HW	0	0	0	0
Other	0	0	0	0
Total Other TCs	***420***	***394***	***8***	***18***

System

Total No. of Test Cases	18754
Over all Total	***41159***

Figure 5.9: Test case results statistic.

Moving to Figure 5.10, which shows how the result of component metrics presents in an grid view format. More precisely, it shows the result of computing component metrics, described in section4.2, where every row of the grid represent one component and the columns show component id, name, version number, number of the component passed test cases, number of the component failed test cases, number of the component not completed test cases, percentage of failed test cases (equation 6.1) and the rest of the columns show the result of the metrics, i.e., number of in port, number of out port, and component interaction.



Window1

Select System by ID * x[redacted]

System Name	System X	Total No.OK TCs	16073
System Version	(10)	Total No.NOK TCs	980
Total No. of Test Cases	18754	Total No.N/A TCs	1701

Metrics Categories: System Metrics, Component Metrics, Requirement Metrics, Testing Metrics, Traceability Metrics, General Metrics

Component Metrics

- ☐ Component Interaction
- ☐ Actual Interactions
- ☐ Total interactions performed
- ☐ Weight of Requirements
- ☐ Weight of In port
- ☐ Weight of out port
- ☐ **Get components outside the current system**

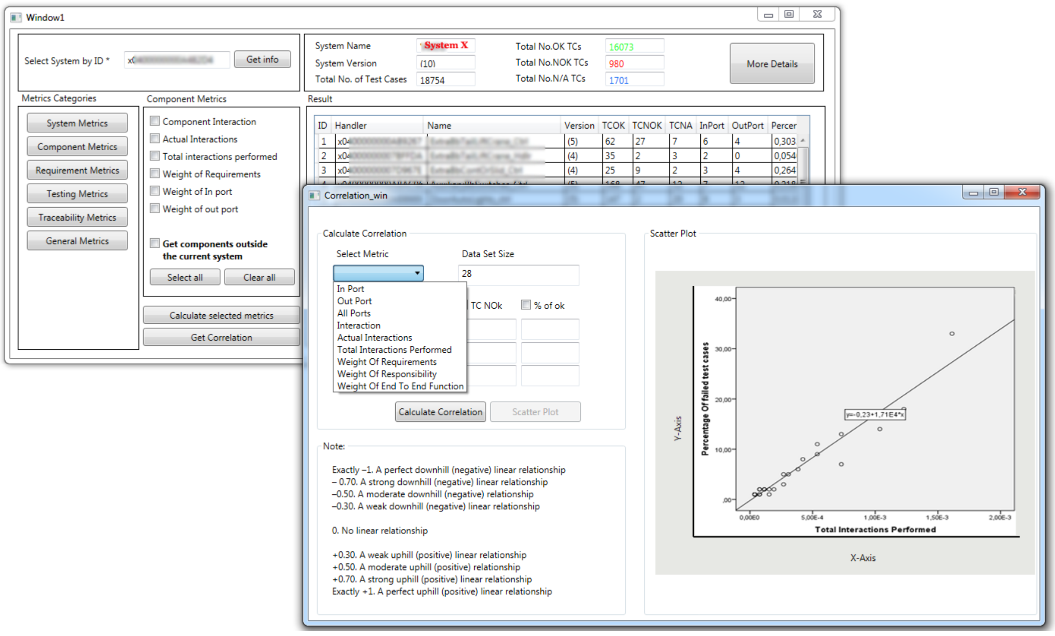
Select all | Clear all | Calculate selected metrics | Get Correlation

Result

ID	Handler	Name	Version	TCOK	TCNOK	TCNA	InPort	OutPort	PercentageOfNOk	TotalPorts	Interaction	ActualInter
1	x0400[redacted]	[redacted]	(5)	62	27	7	6	4	0,303370786516854	10	1,5	0,05376344
2	x0400[redacted]	[redacted]	(4)	35	2	3	2	0	0,0540540540540541	2	0	0,01075269
3	x0400[redacted]	[redacted]	(4)	25	9	2	3	4	0,264705882352941	7	0,75	0,03763441
4	x0400[redacted]	[redacted]	(5)	168	47	12	7	12	0,218604651162791	19	0,5833333	0,1021505
5	x0400[redacted]	[redacted]	(5)	147	2	20	8	3	0,0134228187919463	11	2,666667	0,05913978
6	x0400[redacted]	[redacted]	(6)	13	0	2	1	1	0	2	1	0,01075269
7	x0400[redacted]	[redacted]	(1)	48	6	0	2	1	0,111111111111111	3	2	0,01612903
8	x0400[redacted]	[redacted]	(1)	63	6	0	1	0	0,0869565217391304	1	0	0,00537634
9	x0400[redacted]	[redacted]	(4)	35	6	0	1	0	0,146341463414634	1	0	0,00537634
10	x0400[redacted]	[redacted]	(5a1)	11	0	2	1	3	0	4	0,3333333	0,02150538
11	x0400[redacted]	[redacted]	(5)	11	0	2	1	0	0	1	0	0,00537634
12	x0400[redacted]	[redacted]	(5)	11	0	2	1	0	0	1	0	0,00537634
13	x0400[redacted]	[redacted]	(3)	49	1	2	2	1	0,02	3	2	0,01612903
14	x0400[redacted]	[redacted]	(3)	50	0	5	2	2	0	4	1	0,02150538
15	x0400[redacted]	[redacted]	(4)	141	5	0	11	3	0,0342465753424658	14	3,666667	0,07526882
16	x0400[redacted]	[redacted]	(6)	32	10	6	9	5	0,238095238095238	14	1,8	0,07526882
17	x0400[redacted]	[redacted]	(7)	65	0	25	33	9	0	42	3,666667	0,2258064
18	x0400[redacted]	[redacted]	(4)	104	0	0	2	1	0	3	2	0,01612903
19	x0400[redacted]	[redacted]	(5)	61	1	4	5	3	0,0161290322580645	8	1,666667	0,04301075
20	x0400[redacted]	[redacted]	(4)	34	1	6	5	2	0,0285714285714286	7	2,5	0,03763441
21	x0400[redacted]	[redacted]	(8)	77	3	20	13	6	0,0375	19	2,166667	0,1021505
22	x0400[redacted]	[redacted]	(5)	29	0	0	1	0	0	1	0	0,00537634
23	x0400[redacted]	[redacted]	(5)	27	1	0	1	0	0,0357142857142857	1	0	0,00537634
24	x0400[redacted]	[redacted]	(9)	24	0	4	18	14	0	32	1,285714	0,172043
25	x0400[redacted]	[redacted]	(7)	99	1	1	2	0	0,01	2	0	0,01075269
26	x0400[redacted]	[redacted]	(10)	13	2	1	14	13	0,133333333333333	27	1,076923	0,1451613
27	x0400[redacted]	[redacted]	(6)	38	2	8	2	3	0,05	5	0,6666667	0,02688172
28	x0400[redacted]	[redacted]	(4a1)	7	0	0	1	1	0	2	1	0,01075269

Figure 5.10: Tool implementation for component metrics.

Finally, Figure 5.11 shows part of the tool statistical functionality. It presents an example for one of a nonparametric measure of statistical dependence between two variables (metrics result and percentage of failed test cases). User can select a metric to calculate it's correlation with test case result, and even generate a Scatter plots to show how much one variable (metrics result) is affected by another (test case result).


Window1

Select System by ID * x0[redacted] Get info

System Name: System X
System Version: (10)
Total No. of Test Cases: 18754
Total No.OK TCs: 16073
Total No.NOK TCs: 980
Total No.N/A TCs: 1701

Metrics Categories: System Metrics, Component Metrics, Requirement Metrics, Testing Metrics, Traceability Metrics, General Metrics

Component Metrics:
- Component Interaction
- Actual Interactions
- Total interactions performed
- Weight of Requirements
- Weight of In port
- Weight of out port
- **Get components outside the current system**

Result

ID	Handler	Name	Version	TCOK	TCNOK	TCNA	InPort	OutPort	Percer
1	x04[redacted]	[redacted]	(5)	62	27	7	6	4	0,303
2	x04[redacted]	[redacted]	(4)	35	2	3	2	0	0,054
3	x04[redacted]	[redacted]	(4)	25	9	2	3	4	0,264

Correlation_win

Calculate Correlation
Select Metric: In Port, Out Port, All Ports, Interaction, Actual Interactions, Total Interactions Performed, Weight Of Requirements, Weight Of Responsibility, Weight Of End To End Function
Data Set Size: 28

Note:

Exactly –1. A perfect downhill (negative) linear relationship
– 0.70. A strong downhill (negative) linear relationship
–0.50. A moderate downhill (negative) relationship
–0.30. A weak downhill (negative) linear relationship

0. No linear relationship

+0.30. A weak uphill (positive) linear relationship
+0.50. A moderate uphill (positive) relationship
+0.70. A strong uphill (positive) linear relationship
Exactly +1. A perfect uphill (positive) linear relationship

Figure 5.11: Tool implementation for the correlation between metrics and test case results

5.3.1 Implementation Validation

The Implementation stage was one of the most critical stage in this project, since any wrong assumption could drive a incorrect conclusion. Especially with a very large data size, which came from different source. To ensure that our implementation and assumption are accurate we followed several procedures, which are:

- Manual checks by comparing the tool result with the raw data using SystemWeaver's views and graphs.
- Xpath checks by applying Xpath queries on the same data and compering its result with the tool result.
- SPSS checks to validate the tool statistical part by comparing the tool statistical result with SPSS result for the same data.
- Expert checks by interviewing Systemite application engineers, as well as expert user from customer side.

Consequently, we modified and improve the tool based on feedback and recommendation from the validation team.

6
Results

This chapter summarizes the outcome of performing the case study, by presenting and interpreting the measurement result, which is crucial for understanding the impact of design attributes in generating faults. First, the data collection process presented in Section 6.1. Second, descriptive statistics described in Section 6.2 .Then, inferential statistics described in Section 6.3. Finally, the data analysis discussed in Section 6.4.

6.1 Data Collection

In this process we collect and measure information on targeted variables, in order to to answer the relevant research questions and hypothesis, and evaluate the outcomes. The study domain is automotive industry, we examine three raw data provided by two Swedish automotive companies and stored in SystemWeaver. The process of data collection divided into two rounds. To move to the second round, the data should meet certain criteria. In the first round we measure the following attributes:

1. Size of the data, i.e., system size (number of requirement, component, etc.)
2. Availability of test case result.
3. Availability of traceability data.

To do that we used XPath expressions [6], the result of this round was to exclude two data out of three, and include one data which contains 40 versions of the same system, that have been developed over more than five years. In the second round, farther filtration was applied, based on the following criteria:

1. The completeness of traceability data, as described in Section5.1.1 (Figure 5.3).
2. The coverage of test cases (the number of test cases applied on a system).

The final result was 12 versions of the same system, these versions had tested both system requirement and component, except one version that tested only system requirement. Since, we used 12 versions to validate the requirement metrics category, and 11 for component metrics category. The overall characteristics of all versions are the fallowing :

- Average number of components = 144 Components.
- Average number of requirements = 245 Requirements.
- Average number of test cases = 23127 Test cases.
  - Average number of passed test cases = 14858 Test cases.
  - Average number of failed test cases = 719 Test cases.

6.2 Descriptive Statistics

This section summarize and describe the evaluable data, which has been collected from a historical record of a Swedish automotive company and stored in SystemWeaver. To extract the data and implement our measurements we built SmartTrace, this tool uses the data available at SystemWeaver as an input and measures the system design attributes as an output, with the ability to visualise and analyse the result. The rest of this section is organised as following, Section 6.2.1 presents the normality check, and Section 6.2.2 presents the Linearity check. Section 6.2.3 describes the correlation between design metrics and the failed test cases. Finally, Section 6.2.4 presents the deference between the mean for the failed and passed test cases with respect to the applied measurements.

6.2.1 Normality Test

Normality is one of the most common assumption made in the development and use of statistical technique [1]. In parametric testing the underlying assumption of normal data is required, Unlike the non parametric testing [2]. There are several methods to evaluate the normality, i.e., numerically and graphically. We applied the graphical method by visualising the destitution of the data using histogram and Boxplot charts. Figure 6.1 shows two example for the nature of the data that we have, it shows the results of two metrics (number of component in port and out port metrics).

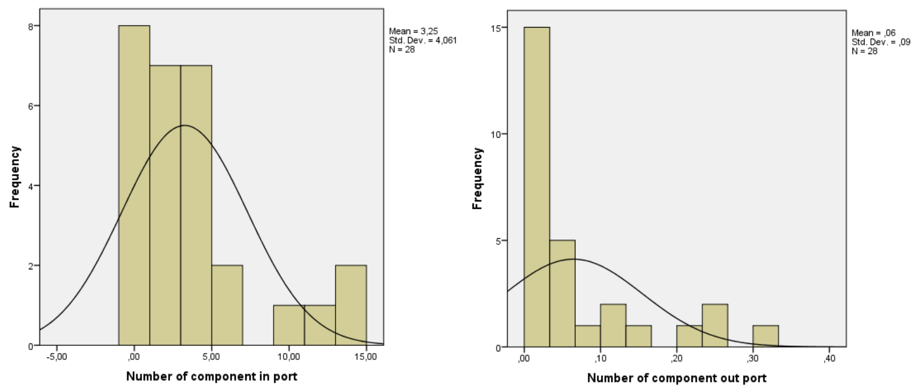


Figure 6.1: Histogram of in ports and out ports metrics data

In general our data was not normally distributed, this affects our decision for selecting the type of the correlation test to be use. Accordingly, non parametric testing was selected.

6.2.2 Linearity Test

Testing the linearity is required because regression, correlation, and other parts of the general liner model assume liner relation between the dependent and non-dependent variables [3]. There are many methods to assess the linearity, for example, graphically and numerically. For our data we used both the graphical and descriptive methods. Figure 6.2 presents an example for the graphical linearity test (Scatter Plot) illustrate the relation between the total interaction performed metrics and the percentage of failed test cases.

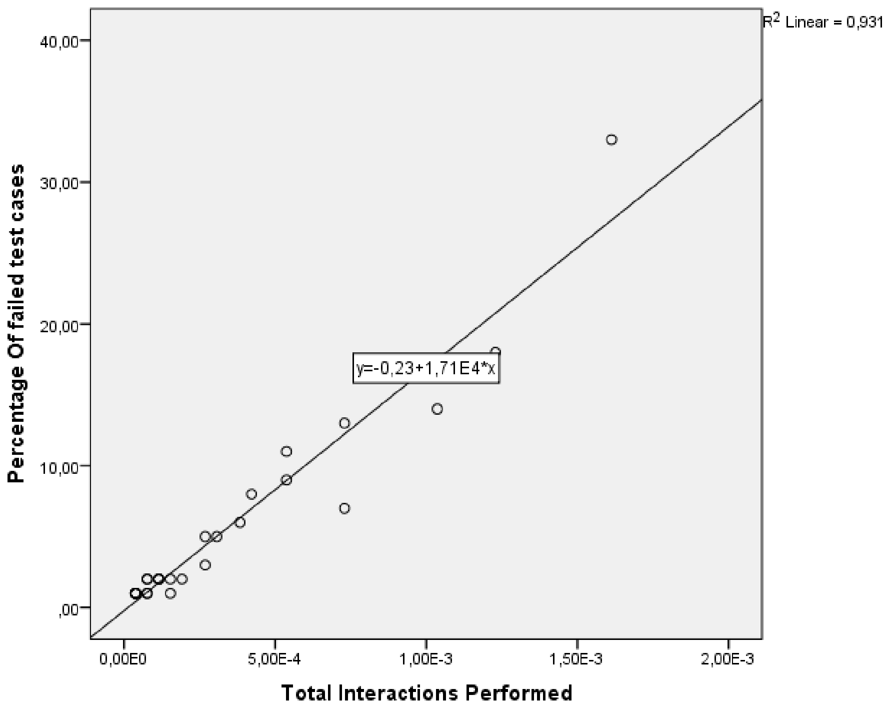


Figure 6.2: Scatter Plot for the total interaction performed metrics and the percentage of failed test cases.

Generally speaking, our data did not show a linear relation between the applied metrics and the failed test cases. Accordingly, non parametric testing was selected.

6.2.3 Spearman's Correlation

Spearman's correlation is a nonparametric, distribution-free, rank statistic proposed by Charles Spearman as a measure of the intensity of an association between two variables [4]. It is applicable when Pearson's assumption are violated. The Pearson's assumption are both variables must be normally distributed, random selection of the sample, the observations are independent, the relation between the two variables is linear, and the variance is constant over the whole data [5]. As mentioned in the previous two subsection, our data violates the Pearson's assumption, which make Spearman's correlation is best choose to be use. Our metrics had divided into four categories, which are system, component, requirement, and testing. The following three tables show the Spearman's correlation between each metrics category and the percentage of failed test cases. The percentage of failed test cases (RerFTC) is defined as following:

$$RerFTC = \frac{NumberofFailedTestCases}{TotalNumberofPerformedTestCases} \tag{6.1}$$

We divide the correlation degree into three levels weak, medium, and strong as shown in Table 6.1 [58], and apply Spearman's correlation on the three metrics categories,which are system, component, and requirement metrics as described in the following subsections.

	Weak	Medium	Strong
ρ	0.2-0.4	0.4-0.6	0.6-1.0

Table 6.1: Correlation degree categories

6.2.3.1 Correlation at System level

In system category we applied five metrics, which are size, cyclomatic complexity, number of functions, number of requirements, and number of edge, as described on Chapter 4 (Section 4.1). Our intention is to investigate the relation between the previous metrics and system fault. In this category, we had two problems, first the sample size, which is 12, second the selection of the sample was not random, where all samples are versions of the same system. Accordingly, we could not drive any conclusion based on the available data. But, still this data could be helpful for future investigation, where more raw data could be include. On the other hand, this data can be use to investigate the relation between the system faults and changes over time. The following line chart, Figure 6.3, shows how the test case results have been changed over 12 versions, the X axises represent the version number, the Y axises represent the number of test cases, and the lines illustrate how passed, failed, not available, total number, and percentage of failed (Equation 6.1) test cases changed over versions.

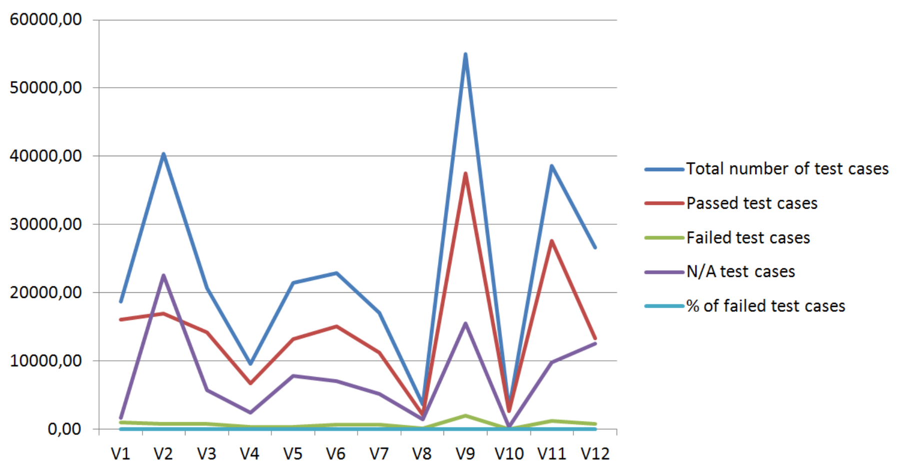


Figure 6.3: Test case results changes over versions

Similarly, Figure 6.4 describes how the test cases result have been changed over 5 years.

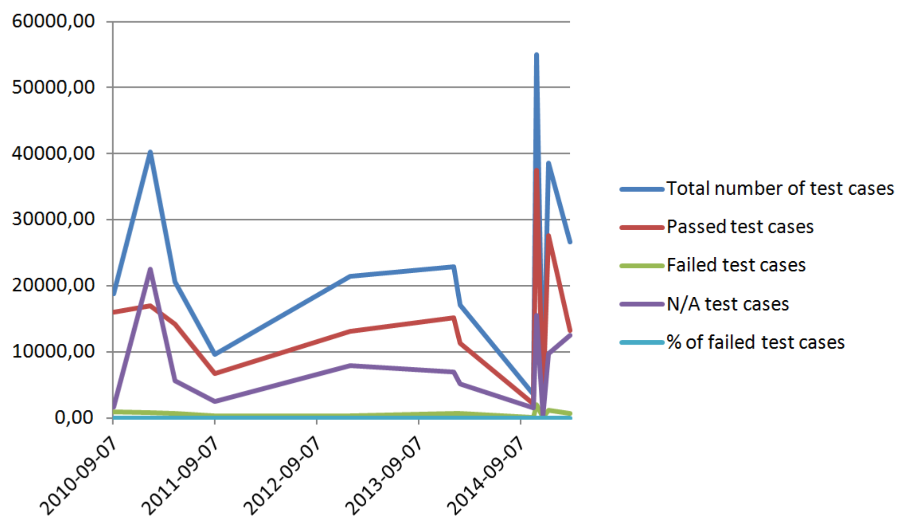


Figure 6.4: Test case results changes over time

6.2.3.2 Correlation at Component Level

In this category we calculated the Spearman's rank correlation coefficient between the percentage of failed test cases and nine of the component metrics, described on Chapter 4 (Section 4.2), only over 11 system versions, since one version out of 12 versions was not tested on component level, as shown in Table 6.2. The horizontal table line represents the system version under test, in this case we have 11 versions encoded v1 to v11. Also, the sample size (number of components) shows under the version number, i.e., N= 28. On the other hand, the vertical table line represents the component metrics. Finally the table cells resulting from intersecting the system versions row and component metrics column shows the correlation result between a component metrics and the percentage of failed test cases in specific system version. For instance, the table cell(1,1) value is 0.842**, that means in system version-1, the Spearman's rank correlation coefficient between the percentage of failed test cases and number of component in port equal to 0.842**. By reference to the correlation degree categories table, Table 6.1, the number of in port metrics shows a strong correlation to the percentage of failed test cases in system version 1. Similarly, the rest of the table cell are filled in.

	Spearman's correlation (ρ)										
Metrics	**V1 N=28**	**V2 N=29**	**V3 N=31**	**V4 N=31**	**V5 N=38**	**V6 N=39**	**V7 N=39**	**V8 N=27**	**V9 N=49**	**V10 N=49**	**V11 N=51**
Number of In Port	,842**	,104	,445*	,102	,227	,322*	,177	,300	,288*	,374**	,112
Number of Out Port	,291	-,096	,393*	-,054	,236	,382*	,230	,159	,356*	,315*	-,086
Total Number of Ports	,943**	-,030	,420*	,003	,237	,361*	,211	,205	,288*	,336*	-,003
Interaction	,715**	,133	,421*	,006	,099	,143	-,004	,162	,070	,204	-,186
Actual Interactions	,943**	-,030	,420*	,003	,237	,361*	,211	,205	,288*	,336*	-,003
Total Interactions Performed	,943**	-,030	,420*	,003	,237	,361*	,211	,205	,288*	,336*	-,003
Number of Requirements	,569**	,016	,655**	,163	,000	,586**	,412**	,261	,275	,280	,133
Number of Responsibility	,732**	,236	,450*	,353	,134	,090	,031	,419*	,126	,047	,065
Number of Function	,643**	,148	,550**	,158	-,044	,410**	,420**	,299	,287*	,334*	,079

Table 6.2: Spearman's rank correlation coefficient between percentage of failed test cases and component metrics.

In short, every component metrics had been tested(correlated to the percentage of failed test cases) over 11 versions of the system, according to our assessment of the correlation degree, shown in Table 6.1, a summary table was built, Table 6.3, shows the number of correlation occurrence over 11 version, based on our estimate of correlation degree. The interpretation of this table along with other analysis parts presented in Section 6.4

	Number of existing Spearman's correlation over 11 system versions			
Metrics	Strong ($\rho = 1.0 - 6.0$)	Medium ($\rho = 0.6 - 0.4$)	Weak ($\rho = 0.4 - 0.2$)	Result (Correlation exist)
Number of In Port	1	1	3	5 /11
Number of Out Port	0	0	3	3 /11
Total Number of Port	1	1	3	5 /11
Interaction	1	1	2	4 /11
Actual Interactions	1	1	3	5 /11
Total Interactions Performed	1	1	3	5 /11
Number of Requirements	1	3	0	4 /11
Number of Responsibility	1	3	0	4 /11
Number of Function	1	3	2	6 /11

Table 6.3: Summary of the Spearman's correlation exists between component metrics and the percentage of failed test cases over 11 system versions.

6.2.3.3 Correlation at Requirement Level

In this category we calculated the Spearman's rank correlation coefficient between the percentage of failed test cases and two of the requirement metrics, Chapter 4 (Section 4.3), over 12 system versions. Similar to what we did in component metrics category in the previous section, the correlation table built, Table 6.4, it shows the relationship between the dependent and independent variables. The horizontal table line represents the system version under test, in this case we have 12 versions encoded v1 to v12. Also, the sample size (number of requirement) shows under the version number. The vertical table line represents the requirement metrics. Finally the table cells resulting from intersecting the system versions row and requirement metrics column shows the correlation result between a requirement metrics and the percentage of failed test cases in specific system version.

	Spearman's correlation (ρ)											
Metrics	**V1 N=97**	**V2 N=108**	**V3 N=116**	**V4 N=94**	**V5 N=152**	**V6 N=199**	**V7 N=197**	**V8 N=104**	**V9 N=228**	**V10 N=27**	**V11 N=49**	**V12 N=243**
Number of Linked Components	-,026	-,026	-,013	-,170	-,076	-,076	-,143*	-,103	-,007	,051	,022	,022
Number of Sub-Requirements	-,082	-,084	-,143	,106	,170*	,168*	,031	,322**	,130	-,014	,132*	,223**

Table 6.4: Spearman's rank correlation coefficient between percentage of failed test cases and requirement metrics

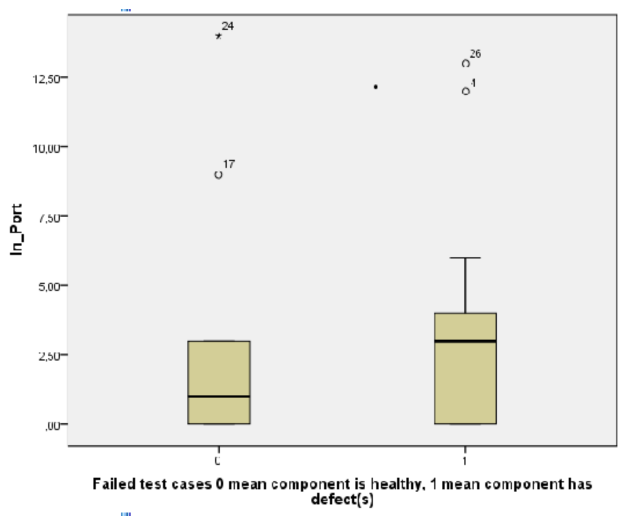
To put it differently and by referring to Table 6.1 a new table created, Table 6.5, to summarize the relation between the percentage of test case failed and requirement metrics. The interpretation of this table along with other analysis parts presented in Section 6.4.

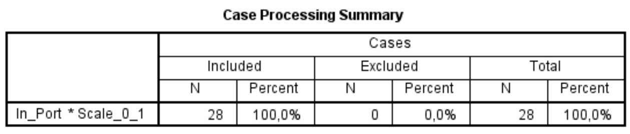
	Number of existing Spearman's correlation over 12 system versions			
Metrics	Strong ($\rho = 1.0 - 0.6$)	Medium ($\rho = 0.6 - 0.4$)	Weak ($\rho = 0.4 - 0.2$)	Result (Correlation exist)
Number of Linked Components	0	0	1	1 /12
Number of Sub-Requirements	0	1	4	5 /12

Table 6.5: Summary of the Spearman's correlation exists between requirement metrics and the percentage of failed test cases over 12 system versions

6.2.4 Mean Difference

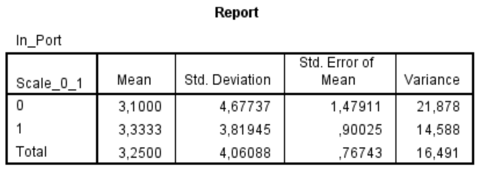
To build a confidence interval, we need to be knowledgeable about the variability of the samples mean differences. For this reason, it is common for a researcher to be interested in the difference between means than in the exact values of the means themselves[7]. In order to visualise the mean of sample, as well as, the difference between means we generate a blot box for every calculated metrics in each system version. For instance, in system version one we generate the blot box related to number of component in port metric, where we compered the means between healthy component(the component that passed all test cases) and the defect component (the component that failed one or more test cases) as shown in Figure 6.5. The figure shows that the mean of the in port number in healthy component equal to 3,100 , where the mean of the in port number in defect component equal to 3,333. Which mean the difference mean between the healthy and defect equal to 0.233. Similarly, the mean difference of the other metrics were computed for the two groups (the healthy and defect artifacts) and reported in the Section 6.4.





Case Processing Summary

	Cases					
	Included		Excluded		Total	
	N	Percent	N	Percent	N	Percent
In_Port * Scale_0_1	28	100,0%	0	0,0%	28	100,0%



Report

In_Port

Scale_0_1	Mean	Std. Deviation	Std. Error of Mean	Variance
0	3,1000	4,67737	1,47911	21,878
1	3,3333	3,81945	,90025	14,588
Total	3,2500	4,06088	,76743	16,491

Figure 6.5: The in port mean difference between the healthy and defected component.

6.3 Inferential Statistics

To generalize our finding, we use inferential statistics to make assessments of the possibility that our observations from the sample data could be correct for the whole population, and they did not happen by chance in this study. For this reason, we applied a non-parametric method, which is Mann–Whitney U test(Wilcoxon rank sum test), provided that the nature of the evaluable data.

6.3.1 Mann–Whitney U Test

It is often hard to have access to large normally distributed samples. Fortunately, there are a bunch of alternative ways to compare two independent groups that do not require perfect normally distributed samples. The MannWhitney U is one of these alternative way tests [38]. In our case, the dependent variables are the metric results, and the independent variables (grouping variables) are the healthy and defect artifacts. We encoded the independent variables as following:

- 0 = Healthy artifact (the artifact passed all test cases)
- 1 = Defect artifact (the artifact failed one or more test case(s))

The test carried out based on the the following assumptions using SPSS and the result reported in Figure 6.6 describes bellow :

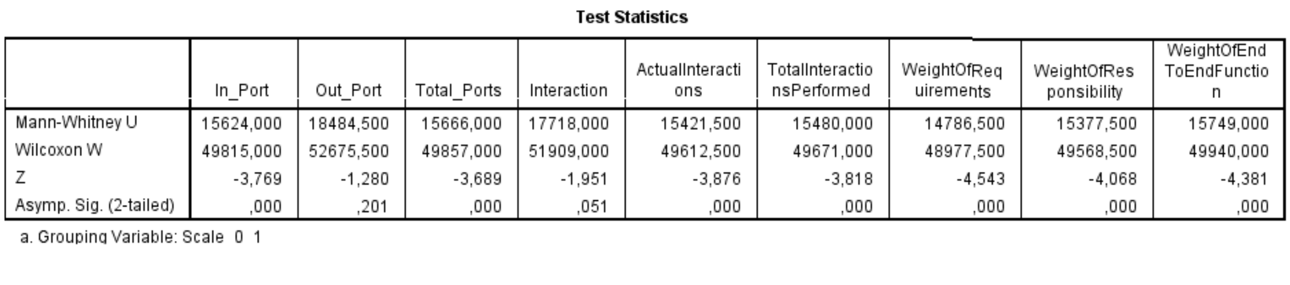
- H0: The distribution of scores for the two groups are equal, in other words, the samples of the both groups came from the same population) [38].
- HA: The distribution of scores for the two groups are not equal [38].
- P-Value = 0.05.

Figure 6.6 shows Mann Whitney U test between two independent variables(grouping variables), which are the healthy and defect component, and dependent variables, which are the component metrics result, with significance levels of 0.05, in a two tail test the following resulted from this test and summarized in Figure 6.7 :

- P-value > 0.05 (retain the null hypothesis)

  - There is no significant difference between healthy and defect component in terms of the number of out ports.
  - There is no significant difference between healthy and defect component in terms of the component interaction.

- P-value < 0.05 (reject the null hypothesis)
  - There is a significant difference between healthy and defect component in terms of the number of in ports.
  - There is a significant difference between healthy and defect component in terms of the total ports number(In and out ports).
  - There is a significant difference between healthy and defect component in terms of the actual interactions.
  - There is a significant difference between healthy and defect component in terms of the total interactions performed.
  - There is a significant difference between healthy and defect component in terms of the Number of requirement.
  - There is a significant difference between healthy and defect component in terms of the Number of responsibility.
  - There is a significant difference between healthy and defect component in terms of the Number of function.

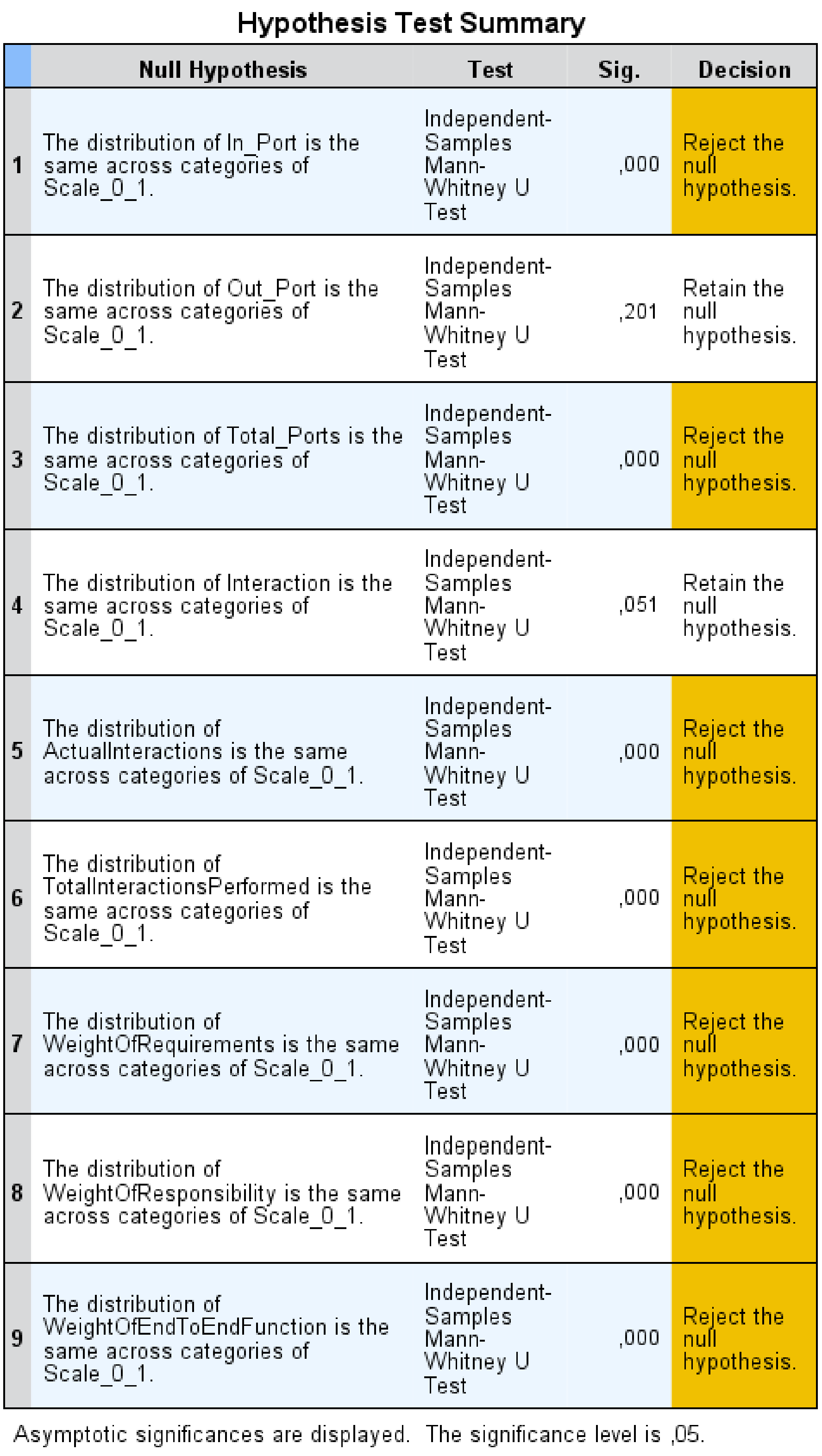


Test Statistics

	In_Port	Out_Port	Total_Ports	Interaction	ActualInteractions	TotalInteractionsPerformed	WeightOfRequirements	WeightOfResponsibility	WeightOfEndToEndFunction
Mann-Whitney U	15624,000	18484,500	15666,000	17718,000	15421,500	15480,000	14786,500	15377,500	15749,000
Wilcoxon W	49815,000	52675,500	49857,000	51909,000	49612,500	49671,000	48977,500	49568,500	49940,000
Z	-3,769	-1,280	-3,689	-1,951	-3,876	-3,818	-4,543	-4,068	-4,381
Asymp. Sig. (2-tailed)	,000	,201	,000	,051	,000	,000	,000	,000	,000

a. Grouping Variable: Scale_0_1

Figure 6.6: Mann Whitney U test between two independent groups(healthy(0) and defect(1) component) and component metrics



Hypothesis Test Summary

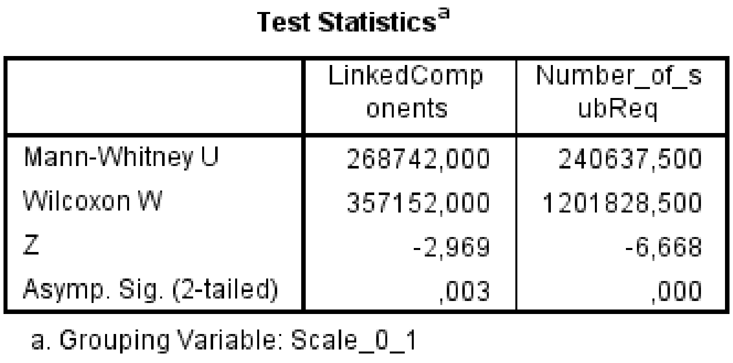
	Null Hypothesis	Test	Sig.	Decision
1	The distribution of In_Port is the same across categories of Scale_0_1.	Independent-Samples Mann-Whitney U Test	,000	Reject the null hypothesis.
2	The distribution of Out_Port is the same across categories of Scale_0_1.	Independent-Samples Mann-Whitney U Test	,201	Retain the null hypothesis.
3	The distribution of Total_Ports is the same across categories of Scale_0_1.	Independent-Samples Mann-Whitney U Test	,000	Reject the null hypothesis.
4	The distribution of Interaction is the same across categories of Scale_0_1.	Independent-Samples Mann-Whitney U Test	,051	Retain the null hypothesis.
5	The distribution of ActualInteractions is the same across categories of Scale_0_1.	Independent-Samples Mann-Whitney U Test	,000	Reject the null hypothesis.
6	The distribution of TotalInteractionsPerformed is the same across categories of Scale_0_1.	Independent-Samples Mann-Whitney U Test	,000	Reject the null hypothesis.
7	The distribution of WeightOfRequirements is the same across categories of Scale_0_1.	Independent-Samples Mann-Whitney U Test	,000	Reject the null hypothesis.
8	The distribution of WeightOfResponsibility is the same across categories of Scale_0_1.	Independent-Samples Mann-Whitney U Test	,000	Reject the null hypothesis.
9	The distribution of WeightOfEndToEndFunction is the same across categories of Scale_0_1.	Independent-Samples Mann-Whitney U Test	,000	Reject the null hypothesis.

Asymptotic significances are displayed. The significance level is ,05.

Figure 6.7: Hypothesis test summary for the two group(healthy(0) and defect(1) component metrics

In like manner to what we did in component metrics, we applied Mann Whitney U test on requirement metrics as well. Coupled with the same assumption of hypothesis, significant level (P = 0.05), and test type(two tail) in component metrics. Using SPSS the test result reported in Figure 6.8. The Figure shows the Mann Whitney U test between two independent variables(grouping variables), which are the healthy and defect requirement, and dependent variables, which are the requirement metrics result. The results were as the following, they are summarized in Figure 6.8 as well: P-value < 0.05 (reject the null hypothesis)

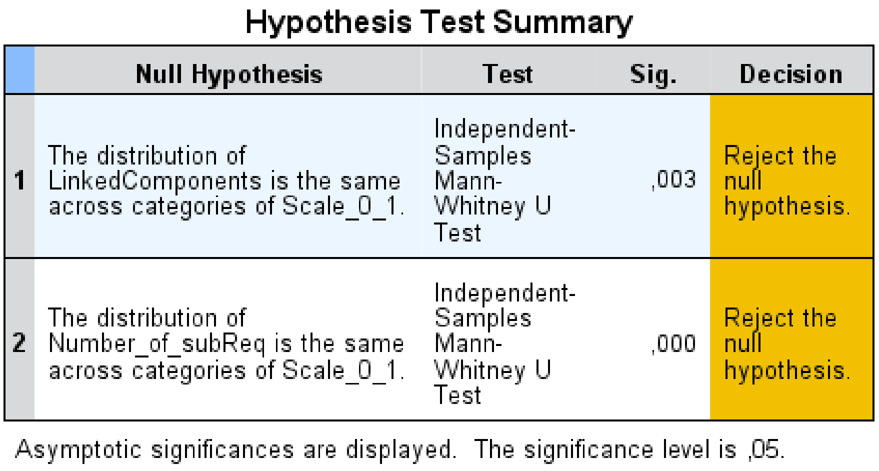
- There is a significant difference between healthy and defect requirement in terms of the number of sub-requirement.
- There is a significant difference between healthy and defect requirement in terms of the number of components linked to a requirement.


Test Statistics[a]

	LinkedComponents	Number_of_subReq
Mann-Whitney U	268742,000	240637,500
Wilcoxon W	357152,000	1201828,500
Z	-2,969	-6,668
Asymp. Sig. (2-tailed)	,003	,000

a. Grouping Variable: Scale_0_1

Figure 6.8: Mann Whitney U test between two independent groups(healthy(0) and defect(1) component) and requirement metrics


Hypothesis Test Summary

	Null Hypothesis	Test	Sig.	Decision
1	The distribution of LinkedComponents is the same across categories of Scale_0_1.	Independent-Samples Mann-Whitney U Test	,003	Reject the null hypothesis.
2	The distribution of Number_of_subReq is the same across categories of Scale_0_1.	Independent-Samples Mann-Whitney U Test	,000	Reject the null hypothesis.

Asymptotic significances are displayed. The significance level is ,05.

Figure 6.9: Hypothesis test summary for the two group(healthy(0) and defect(1) requirement metrics

6.4 Data Analysis

This section reporting and discussing our findings. In this project we investigate whether the relationships between requirements and design artifacts, as well as, information from individual artifacts, can be leveraged to predict software faults. In order to do that we measured different aspects of requirement, component, system, and test cases, taking the advantages of the existing traceability data, which allowed us to to track the faults in both ways, forward and backward, and investigate the cause of the fault in quantitative method. With this intention, we divided the metrics into four categories, as shown in Chapter 4 and investigated each category along with the associated test case suite. By applying both the descriptive statistics and inferential statistics on the available data we reach the following results:

In system category, we had two limitations, in the first place the sample size was only 12 samples, second all these samples are a versions of the same system, which inconsistent with the condition of random sample selection. Under those circumstances, we could not tell if the system design attributes, like a system complexity, size, weigh (number of requirement, number of method, number of edge, etc.) can be leveraged to predict software faults. Consequently, we were unable to prove our first hypothesis, which is " A higher- complexity system design has a higher failure rate.". However, difference in means across the 12 versions of the system gives an indicator that system versions with higher complex design have a higher failure rate. Although this may be true, but still there is no enough statistical evidence to support our claim. Accordingly, more data is needed to validate our hypothesis, which we consider in our future plan, due to unavailability of more data in this thesis.

Moving to component category, we applied nine metrics measure three attributes of a component, which are wight (in port, out port, total port, and requirement), interaction(component interaction, actual interaction,and total interaction performed), and load (number of component responsibility and number of function). Based on the results of the previous statistical tests and the provided data, we found that component design attribute can be leveraged to predict software faults, more precisely, each of the following component attribute shows a correlation to component faults. The degree of the correlation was different among them, so the following order present them from strongest(1) to weakest (3):

1. Number of function.
2. Number of requirement and number of responsibility.
3. Actual interaction, total interaction performed, in port, and total number of port.

Accordingly, we could accept our second hypothesis, which is "Software components that have a higher interaction with other component have a higher chance of failing a test.", as well as, the third hypothesis, which is "Software components that have a higher number of responsibility have a higher chance of failing a test.", together with, the fourth hypothesis, which is "Software components that are responsible for (i.e., linked to) a higher number of requirements have a higher chance of failing a test.". However, non of the previous metrics showed 100% correlation over the 11 version, as described before in Table 6.3. Accordingly, we discussed our finding quality in Chapter 7(Section 7.2).

Finally, the requirement category, in this category we applied two metrics to investigate whether a software requirement linked to large number of components tend to have more test failures than average(hypothesis 5), as well as, whether software requirement that have a higher number of sub-requirements have a higher chance of failing a test(hypothesis 6). The statistical test result did not show a clear relation between a requirement linked to large number of components and a higher chance of failing test. Accordingly, we do not have enough statistical evidence to support our claim, so we failed to prove the hypothesis 5. Moving to hypothesis 6, the statistical test result showed a weak relation between a higher number of sub-requirements and a higher chance of failing test, as shown in Table 6.5. Accordingly, we need to study more data, in order to support our claim.

7
Conclusion

This chapter starts with summarizing the thesis work and listing our finding. Which is followed by a discussion section that describe the quality of finding, benefit, and suggestions for both Systemite and the data provided companies. Finally, future works are presented.

7.1 Summary

Software defect prediction is the process of tracing defective components in software before the start of implementation and testing phases [54]. Having an early predictors is useful and desirable as it can reduce the cost of software development[16] and [17], eliminate the majority of defect since more than that 60% of defects are generated at the early stage of development [16], reduced development time, reduced rework effort, increased customer satisfaction, and improve the reliability and quality of a software [54].
In this project we investigated whether the relationships between requirements and design artifacts, as well as information from individual artifacts, can be leveraged to predict software faults. As a novel contribution, we used information from requirements and design artifacts (and the traceability links between them) to predict faults. More precisely, we examined six hypotheses related to system, component, and requirement design attributes. Theses hypotheses propose that a higher complexity system design, a higher interaction component, a component with higher number of responsibility, a component liked to higher number of requirements, a requirement linked to higher number of components, and a requirement has a higher number of sub-requirements have a higher chance of failing a test.

To answer our research question and prove our hypotheses a quantitative case study conducted together with our industrial partner Systemite AB had had been carried out, with data provided by two Swedish automotive companies. The studied data was stored in SystemWeaver, which is a holistic information management solution for systems engineering and software development, developed by Systemite and has been use by automotive industry from more than 15 years.

After building the partnership and setup the project setting, we review the related work exists in both the academia and industry, as well as our previous project with the same partner, which was the entrance for this thesis. As a result, we select 19 metrics to implement in this work, which measure different aspects of a system and

its artifacts design attributes, taking the advantages of the evaluable traceability data. In order to perform the measurements and present their results, a tool has been implemented which is able to apply the design metrics, analyse the result of the metrics by applying statistical test, and visualized their result.
Then, we validate our result and implementation, by interviewing Systemite's experts and data engineers from the data provider company side, as well as, using SystemWeaver's script language (Xpath) to verify the tool results.

Then, we analyse the data using our tool and SPSS, by applying both the descriptive and inferential statistical, i.e., normality test,linearity test, mean difference, Spearman's correlation, Mann–Whitney U test, etc.

Finally, a conclusion was drawn based on the data given. We found that design metrics and traceability data could be use to detect software fault patterns, which could be use later to build a powerful fault prediction model that support system design engineers decisions. In this project we found the following:

- More than the half of the studied data (6 out of 11 versions) shows relation between component high number of in port and a higher chance of failing a test
- Approximately half of the studied data (5 out of 11 versions) shows a relation between total number of component ports (in ports and out ports) and the percentage of failed test case. While only 3/11 of the studied data shows a weak relation to a higher chance of failing a test.
- Less than the half of the studied data (5 out of 12 versions) shows a relation between a higher number of sub-requirements and a higher chance of failing a test.
- Only 4/11 of the studied data shows a relation between a component liked to higher number of requirements and a higher chance of failing a test.
- Only 3/11 of the studied data shows a relation between a higher number of component responsibilities and a higher chance of failing a test.
- Only 1/12 of the studied data shows a relation between a requirement linked to a higher number of components and a higher chance of failing a test.
- Regarding the system design attribute, like a system size and a design complexity, we failed to prove our claim due to the available data limitation. In the event that, we have a small sample size (12 System), even those sample are not randomly select (all of them versions of the same system).

7.2 Discussion

This section describe the quality of our finding, benefit of this project, and suggestions for both Systemite and the data provided companies.

Starting with the quality of finding, the study data was collected from a mature project developed by one of the largest Swedish automated company. This project has been developed over the last five years, and the development process was managed using SystemWeaver, where all project information is stored. With this data we conducted our study and extracted our finding. However, as listed above non of our finding present 100% support for our claims, i.e., 55% of the studied data support that a high number of component in port may cause a higher chance of failing a test. Of course having more data that support our claims would sharp our finding, but still the current finding could give an indicators about the correlation between the system design attributes and faults. To have a stronger result more data need to be study, which we planed to do in our future work, even from different industries.

Moving to the benefits obtained from this project. As a novel contribution, we used information from requirements and design artifacts, and the traceability links between them, to predict faults. Accordingly, we extract several fault patterns, based on the provided data. These patterns could be use later to predict fault in future system releases or other system. The advantages of those patterns that they are retrieved from the same related data, which make them more accurate predictor, comparing to patterns retrieved from other industry and applied on automotive industry. For example, a fault prediction model build based on data provided by space industry (NASA)[55], to be use later to predict fault on other industries. Another benefit for this work, it highlights the advantages and important of creating the traceability data. For example the studied data was provided by two companies, we failed to study two data, because these data did not have a traceability data or it is not complete. Equally important, the traceability data can be use for more analyses not only for fault prediction, i.e., change management. Finally, from technical perspective, this project proposed a new technique to define design metrics using Xpath [6], as well as it suggested a fault prediction model [39]. A primary version of this model was developed in this thesis, as described in 5.3.

Finally, as a result of this project we have some suggestions for both our partner (Systemite) and their customer (the data provided companies).
First, starting with Systemite, we recommend them to automate, or semi-automate, the process of creation the traceability links, i.e., if a user create a test case to test a requirement, then the platform automatically create a link between the test case and the system, that contain the tested requirement.

Second, in some cases the reason behind missing the traceability data is using different tool to manage the development process, we highly recommend both Systemite and the data provided company to improve the communication and collaboration between tools. Taking in consideration that both companies already recognize this need and done some related project, but still more work could improve the system development process, decrease the development complexity, and improve the collaboration between connected companies.

Third, another reason for missing the traceability data, even when the tool support that, is human factor, since the system development process done by different teams, i.e., system designer, developer, and tester, in some cases the traceability between the work of those teams is missing, i.e., the traceability between system architecture and testing. We highly recommend to raise the awareness about the important of creating the traceability data, even if it is sometimes consuming resources, and did not show a instantly benefits.

In short, the traceability data was a fundamental part of this project, it allows us to study the whole chain of the development process, starting from system design to testing, by enabling us to trace all system artifices in both forward and backward direction. Software fault prediction is only one use of traceability data, since it could be usefully for many other uses. Accordingly, we highly recommend to create complete traceability data. Moreover, predicting faults at early stage of development process could improve the whole development process in significant manner, accordingly more work need to be done in this regard.

7.3 Future Works

This study constituted an early prediction of software faults using traceability data and design metrics. In order to expand this work and achieve our ultimate goal to develop a fault prediction and decision support system, a number of approaches are suggested. First, expand the case study to cover other industries, as describes in Section 7.3.1. Second, define and implement more design metrics, as presents in Section 7.3.2. Third, study the relationship between system changes and generate faults, as describes in Section 7.3.3. Finally, improve the current version of our tool, which we developed in this thesis, to become a fully automated fault prediction model and decision support system, as suggests in Section 7.3.4 .

7.3.1 Study Other Industries

One of the thesis's limitations is that the studied data came only from companies working in automotive industry. Accordingly, our finding could not be generalized to other industries, as mentioned before in Section 3.3.2. To come over this limitation and generalize our finding, we plan to study other industries, i.e., banking and telecommunication, even expand our study within the automotive industry by studying other automotive companies. Having more data to study will offers several advantages.

First, it will sharp our finding, by increasing the data size. Second, it will add a new dimension for our study, this dimension is to compare different industries in term of generating fault, since every industry has its own features . For instance, automotive systems should follow several standard and regulation, i.e., ISO 26262, whereas bunking system should meet a certain level of security. Third, it will allow us to generalize our finding and discover more fault patterns.

7.3.2 Define More Metrics

In this thesis work we implement 19 metrics for the purpose of measuring system, component, and requirement design attributes. The selection of theses metrics was based on literature, properties of the studied data, and Systemite's previous projects.

Having more data from different sources will allow us to define more metrics. Part of our future plan is to implement more metrics recommended by the software community, which we could not implement it in this thesis, because of the limitations of the available data and resource for this project. For instance, all metrics that we implemented are a quantitative metrics, where we used the traceability data to measure a system quantitative attributes, i.e., number of components used (linked to) requirement or number of sub-requirement per requirement. For the future we plan to implement qualitative metrics, i.e., number of requirement weak phrase or requirement ambiguity[53]. To do that we need to implement textual analysis in our model to be able to interpret the characteristics of a requirement textual description.

7.3.3 System Changes and Faults

Another part of or future plan is to investigate the relationship between system changes and generate faults, which will improve the prediction power of our fault prediction model. The studied data showed fluctuations in the percentage of failed test cases (Equation 6.1), as well as the number of failed and passed test case over 12 versions of the system, which cover a time period of approximately 5 years, as shown in Figure 7.1. However, this data alone could not be enough to be study and draw strong conclusion for two reasons. First, the sample size is only 12. Second, these 12 version are selected out of 40 versions, because they had tested by the data provided company, accordingly we got access to their test case results. Since, the other versions were tested by other automotive supplies companies. Having more data to study will allow us to investigate this relationship more, as well as advance our model.

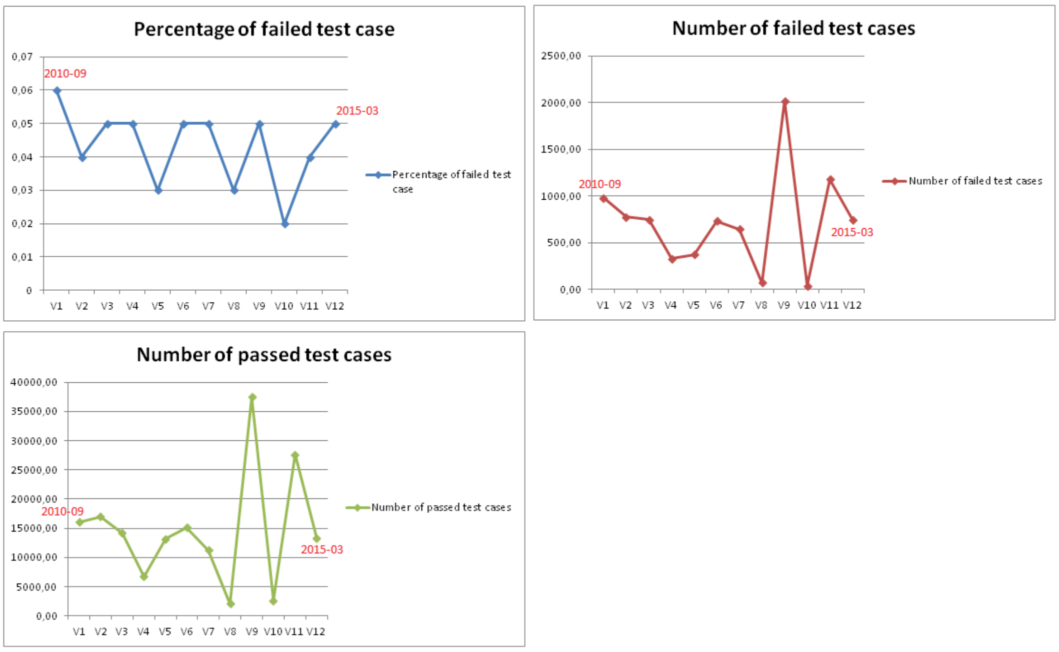


Figure 7.1: The relationship between system changes and generate faults.

7.3.4 Improve SmartTrace

Our ultimate goal is to build a fault prediction model, that predict system faults at the early stage of development process, which guides the engineers towards the identification of problematic components and requirements. This master thesis work define the basic blocks to build this model, which we called SmartTrace and reported in [39]. As a part of our future plan we aim to expand the current version of the tool that we implemented in this thesis, which described in Chapter 5, to build a fault prediction and decision support system (SmartTrace).

SmartTrace is designed to mature continuously as the product development data in the organization grows. The data, often created by users, is used to generate insights on the patterns that are likely to result in failures. These insights are used to provide the user with decision support as shown in Figure 7.2 [39].

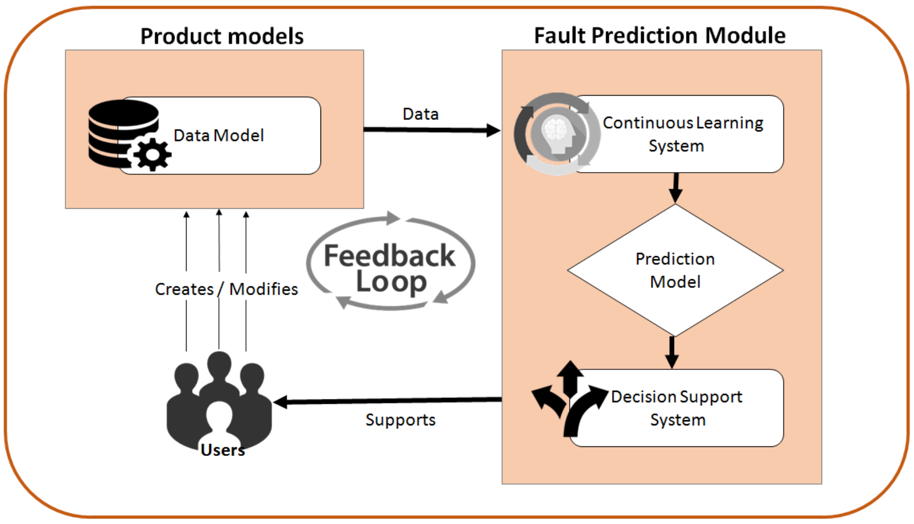


Figure 7.2: The SmartTrace fault prediction model using the traceability data in the early stage of development life cycle.

Figure 7.2 shows how SmartTrace intended to works, system's designers trigger the model by creating or modifying the product data. The traceability data between the product artifacts allows us to investigate the fault patterns, by applying statistical methods over the product data. We use the traceability between the product artifacts in addition to the isolated artifacts to retrieve these patterns.

The first part of the module is a continuous learning system, which grows organically and continuously to update retrieved data from the product models, by adding the experience gained from new iteration to the old data. Over time the learning system will be more capable to provide the prediction module with necessary data to discover more potential faults.

Moving to the second part, which is the prediction model, it is responsible for predicting faults based on existing fault patterns (this thesis aimed to define some of those patterns) historical product data, and traceability data. The prediction module uses this data to find the faults or point to the places where the potential faults could be hidden. The last part of this module is the decision support system, which provides the users with real time decision support. An essential part of this module deals with extracting the fault patterns. The patterns extraction process starts by defining and calculating metrics, these metrics measure a wide variety of product attributes, as mentioned before. The next step is to find the correlation between the calculated metrics and the related test cases, we use standard statistical tools, like non-parametric tests (e.g., Wilcoxon). A significant correlation with test failures means that the metrics can be used to identify the product elements related to the metrics (e.g., an architectural component) as a source of faults.

The decision support system recognizes these occurrences and check the product based on the available patterns. Accordingly, the decision support system warns the developers about the potential risk and provides them with suitable suggestions (e.g., how to refactor the system to improve on the relevant metrics). For instance, if the number of dependencies shows a likelihood of introducing failures in one component, the decision support can suggest to split the component into two smaller components or to restructure the allocation of software components onto the electronic control units (ECUs).

Bibliography

[1] Kiefer, N.M. and Salmon, M., 1983. Testing normality in econometric models. Economics Letters, 11(1-2), pp.123-127.

[2] Gibbons, J.D. and Chakraborti, S., 2011. Nonparametric statistical inference (pp. 977-979). Springer Berlin Heidelberg.

[3] Garson, G.D., 2012. Testing statistical assumptions. Asheboro, NC: Statistical Associates Publishing.

[4] Hauke, J. and Kossowski, T., 2011. Comparison of values of Pearson's and Spearman's correlation coefficients on the same sets of data. Quaestiones geographicae, 30(2), pp.87-93.

[5] Bachman, L.F., 2004. Statistical analyses for language assessment. Ernst Klett Sprachen.

[6] Soderberg, J., Shahrokni, A. and Nassar, B., 2016. Using XPath to Define Design Metrics. International Conference on Performance, Safety and Robustness in Complex Systems and Applications (PESARO). IARIA, 2016.

[7] Lane, D.M., 2015. Online statistics education: An interactive multimedia course of study.

[8] Jiang, Y., Cukic, B. and Menzies, T., 2007, November. Fault prediction using early lifecycle data. In Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on (pp. 237-246). IEEE.

[9] Hassan, A.E., 2009, May. Predicting faults using the complexity of code changes. In Proceedings of the 31st International Conference on Software Engineering (pp. 78-88). IEEE Computer Society.

[10] Catal, C., 2011. Software fault prediction: A literature review and current trends. Expert systems with applications, 38(4), pp.4626-4636.

[11] Rathore, S.S. and Gupta, A., 2012, September. Investigating object-oriented design metrics to predict fault-proneness of software modules. In Software Engineering (CONSEG), 2012 CSI Sixth International Conference on (pp. 1-10). IEEE.

[12] Radjenović, D., Heričko, M., Torkar, R. and Živkovič, A., 2013. Software fault prediction metrics: A systematic literature review. Information and Software Technology, 55(8), pp.1397-1418.

[13] Zhou, Y., Xu, B., Leung, H. and Chen, L., 2014. An in-depth study of the potentially confounding effect of class size in fault prediction. ACM Transactions on Software Engineering and Methodology (TOSEM), 23(1), p.10.

[14] Mahajan, R., Gupta, S.K. and Bedi, R.K., 2015. Design of Software Fault Prediction Model Using BR Technique. Procedia Computer Science, 46, pp.849-858.

[15] Singh, P. and Verma, S., 2015. Cross Project Software Fault Prediction at Design Phase. World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering, 9(3), pp.800-805.

[16] Mohan, P., Shankar, A. U., and JayaSriDevi, K, 2012. Quality Flaws: Issues and Challenges in Software Development. Computer Engineering and Intelligent Systems, 3(12), 40-48.

[17] Stecklein, J.M., Dabney, J., Dick, B., Haskins, B., Lovell, R. and Moroney, G., 2004. Error cost escalation through the project life cycle.

[18] Chen, J., Yeap, W.K. and Bruda, S.D., 2009, May. A review of component coupling metrics for component-based development. In World Congress on Software Engineering (pp. 65-69). IEEE.

[19] Jiang, Y., Cuki, B., Menzies, T. and Bartlow, N., 2008, May. Comparing design and code metrics for software quality prediction. In Proceedings of the 4th international workshop on Predictor models in software engineering (pp. 11-18). ACM.

[20] Chidamber, S.R. and Kemerer, C.F., 1994. A metrics suite for object oriented design. Software Engineering, IEEE Transactions on, 20(6), pp.476-493.

[21] Hristov, D., Hummel, O., Huq, M. and Janjic, W., 2012. Structuring software reusability metrics for component-based software development. In Proceedings of Int. Conference on Software Engineering Advances (ICSEA).

[22] McCabe, T.J. and Butler, C.W., 1989. Design complexity measurement and testing. Communications of the ACM, 32(12), pp.1415-1425.

[23] Wikipedia, "Cyclomatic complexity", 2016. [Online]. Available:https:$//en.wikipedia.org/wiki/Cyclomatic_complexity.$ [Accessed: 06- Jan- 2016].

[24] [Wikipedia, "Coupling (computer programming)", 2016. [Online]. Available: https:$//en.wikipedia.org/wiki/Coupling_(computer_programming).[Accessed : 06 - Jan - 2016]$.

[25] Support.objecteering.com, "Coupling Between Object Classes (CBO)", 2016. [Online]. Available: http:$//support.objecteering.com/objecteering6.1/help/us/metri$

[26] Lam, W., Loomes, M. and Shankararaman, V., 1999. Managing requirements change using metrics and action planning. In Software Maintenance and Reengineering, 1999. Proceedings of the Third European Conference on (pp. 122-128). IEEE.

[27] Suma, V. and Nair, T.R., 2012. Defect Management Strategies in Software Development. arXiv preprint arXiv:1209.5573.

[28] Jiang, Y., 2009. Incremental Development and Cost-based Evaluation of Software Fault Prediction Models (Doctoral dissertation, West Virginia University).

[29] Suma, V. and Nair, T.R., 2012. Defect Management Strategies in Software Development. arXiv preprint arXiv:1209.5573.

[30] Singh, P., Verma, S. and Vyas, O.P., 2014. Software Fault Prediction at Design Phase. Journal of Electrical Engineering Technology, 9(5), pp.1739-1745.

[31] Xu, J., Ho, D. and Capretz, L.F., 2008. An empirical validation of object-oriented design metrics for fault prediction.

[32] Aggarwal, K.K., Singh, Y., Kaur, A. and Malhotra, R., 2007. Investigating effect of Design Metrics on Fault Proneness in Object-Oriented Systems.Journal of Object Technology, 6(10), pp.127-141.

[33] Sehgal, R. and Mehrotra, D., 2014. Analysis of Software Fault Prediction Metrics. World Applied Sciences Journal, 32(3), pp.368-378.

[34] Cleland-Huang, J., Gotel, O.C., Huffman Hayes, J., Mäder, P. and Zisman, A., 2014, May. Software traceability: trends and future directions. In Proceedings of the on Future of Software Engineering (pp. 55-69). ACM.

[35] Cleland-Huang, J., Gotel, O. and Zisman, A., 2012. Software and systems traceability (Vol. 2, No. 3, pp. 7-8). Springer.

[36] Hayes, J.H., 2003, November. Building a requirement fault taxonomy: Experiences from a NASA verification and validation research project. In Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on (pp. 49-59). IEEE.

[37] Malhotra, R., 2015. A systematic review of machine learning techniques for software fault prediction. Applied Soft Computing, 27, pp.504-518.

[38] Nachar, N., 2008. The Mann-Whitney U: A test for assessing whether two independent samples come from the same distribution. Tutorials in Quantitative Methods for Psychology, 4(1), pp.13-20.

[39] Nassar, B., Shahrokni, A. and Scandariato, R., 2016. Traceability Data in Early Development Phases as an Enabler for Decision Support.International Workshop on Emerging Trends in DevOps and Infrastructure, XP, 2016.

[40] Glass, R.L., 1994. The software-research crisis. IEEE Software, 11(6), p.42. Vancouver

[41] Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El Emam, K. and Rosenberg, J., 2002. Preliminary guidelines for empirical research in software engineering. Software Engineering, IEEE Transactions on, 28(8), pp.721-734.

[42] Creswell, J.W., 2013. Research design: Qualitative, quantitative, and mixed methods approaches. Sage publications.

[43] Systemweaver.se, (2016). SystemWeaver - Managing product development information:: Home. [online] Available at: http://www.systemweaver.se [Accessed 14 Feb. 2016].

[44] Systemite.se, (2016). Systemite AB | Keeping complex systems consistent, correct and complete.. [online] Available at: http://www.systemite.se [Accessed 15 Feb. 2016].

[45] Runeson, P. and Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. Empirical software engineering, 14(2), pp.131-164.

[46] Ridley, D., 2012. The literature review: A step-by-step guide for students. Sage.

[47] Levy, Y. and Ellis, T.J., 2006. A systems approach to conduct an effective literature review in support of information systems research. Informing Science: International Journal of an Emerging Transdiscipline, 9(1), pp.181-212.

[48] synligare.Visibler Development of Embedded Automotive Systems - Visualised Requirements Management and Collaboration.

(2015). Grant FFI 2013-01296. [online] Sweden. Available at: http://synligare.eu/deliverables/Synligare$_{D}eliverable_{D}1.1_{V}1.2.pdf[Accessed18May2016].Syste$

[49] SystemWeaver software. (2016). Gothenburg, Sweden: Systemite AB.

[50] W3.org. (2016). XML Path Language (XPath) 2.0 (Second Edition). [online] Available at: https://www.w3.org/TR/xpath20/ [Accessed 20 May 2016].

[51] Mathdotnet.com. (2016). Math.NET. [online] Available at: http://www.mathdotnet.com/ [Accessed 23 May 2016].

[52] Runeson, P., Host, M., Rainer, A. and Regnell, B., 2012. Case study research in software engineering: Guidelines and examples. John Wiley Sons.

[53] Wohlin, C. ed., 2005. Engineering and managing software requirements. Springer Science Business Media.

[54] Rawat, M.S. and Dubey, S.K., 2012. Software defect prediction models for quality improvement: a literature study. IJCSI International Journal of Computer Science Issues, 9(5), pp.288-296.

[55] L. Guo, Y. Ma, B. Cukic, H. Singh, "Robust prediction of fault proneness by random forests," In: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE04), pp. 417–428, 2004.

[56] Synligare.eu. (2016). Synligare Home. [online] Available at: http://synligare.eu/HomePage.html [Accessed 30 May 2016].

[57] Azram, N.A. and Atan, R., 2012. Traceability Method for Software Engineering Documentation. International Journal of Computer Science Issues(IJCSI), 9(2).

[58] Bmj.com. (2016). 11. Correlation and regression | The BMJ. [online] Available at: http://www.bmj.com/about-bmj/resources-readers/publications/statistics-square-one/11-correlation-and-regression [Accessed 9 Jun. 2016].

A
Appendix 1

Using XPath to Define Design Metrics
PESARO 2016 : The Sixth International Conference on Performance, Safety and Robustness in Complex Systems and Applications. Lisbon, Portugal

Using XPath to Define Design Metrics

Jan Söderberg, Ali Shahrokni
Systemite AB
Gothenburg, Sweden
e-mail: {jan.soderberg, ali.shahrokni}@systemite.se

Bashar Nassar
Chalmers University of Technology
Gothenburg, Sweden
e-mail: bashar.h.nassar@gmail.com

***Abstract*—Architecture description formats like EAST-ADL and automotive open system architecture (AUTOSAR) use an extensible markup language (XML) based file representation. The complexity of the systems based on these architecture description languages often call for metrics definitions for the purpose of complexity or completeness management. The Swedish research project *Synligare* deals with improved management of complex systems based on EAST-ADL. One result from the project was that XPath could be used as a basis for the definition of design metrics, offering several advantages. XPath has further been demonstrated in the project to offer sufficient expressiveness and usability for the purpose.**

Keywords-Metrics; XPath; EAST-ADL; AUTOSAR; Exchange metrics.

I. INTRODUCTION

The evolution rate of automotive electric/electronic(E/E) systems has increased exponentially during the last decade, and the number of electronic control units now typically amounts to 50-100 [1].New and complex functionalities and technologies are emerging, making the prospect of autonomous driving within reach [2]. A consequence of the higher complexity is that the classical document and file based methods are no longer sufficient to manage the product and process data. We have seen that the Software specification of a single Electronic Control Unit (ECU)can be in excess of 8.000 pages. Meanwhile, there is an increased demand for reduced development cycles and product costs.

Synligare[1] is a Swedish industrial research project that aims to improve methods and tool support for model-based development of automotive E/E systems within and between organizations [7]. The members of the Synligare project include Volvo AB, ArcCore AB, Autoliv AB, Semcon and Systemite AB. The parties represent the different roles in a typical E/E development project, including Volvo as a manufacturer and integrator ("OEM" in current automotive terminology), Autoliv as a Tier 1 supplier, ArcCore as a Tier 2 supplier, Systemite as a high level modeling tool supplier on high levels of abstraction, ArcCore as low level modeling tool supplier, and Semcon as a specialist engineering service supplier.

The project uses the EAST-ADL language [8] as a common specification for exchanging developed data within and between organizations. EAST-ADL is an adaptation of SysML[9] for automotive E/E systems. The language includes support for high level specifications of the system, for instance, vehicle features, down to the implementation level, based on AUTOSAR[10]. The language includes optional packages for modeling of variability, timing, safety, and more.

One of the main objectives of the Synligare project is to enable exchange of functional safety data inside and across organizations. ISO 26262 is a standard for functional safety that challenges the automotive industry. The data is produced on different location by different companies. However, the progress needs to be measured, updated, and consolidated in different companies and exchanged between suppliers and OEMs. Many process and products metrics in the ISO 26262 standard are valid across organization boundaries. Many of the progress metrics can be extracted from product data. For instance, one such metric is the state of progress of the verification process for all technical safety requirements, or the state of fulfillment of safety goals on different levels of abstractions.

The Synligare project specifically addresses data exchange challenges between OEMs and suppliers. When the exchange is based on a single formalized representation like EAST-ADL the efficiency and quality of the exchange can be significantly improved, since handover of development, tracing impact of changes and analysis of data can be automated.

A remaining challenge when information is shared and exchanged is to assure that all involved parties can interpret the information in the same way. Although the XML based exchange format for EAST-ADL provides a formalization of the information, the way this information is viewed by different parties is not specified; specifically, when it comes to design metrics. For instance, EAST-ADL does not include progress measurements such as completeness or complexity of the design. In the Synligare project, these metrics were originally specified in natural language, with references to the constructs of the language. For specifying the metrics, we used a more formal alternative, inspired by XPath expressions[11], to express the metrics. These metrics could then be shared between different tools at the OEM and supplier sides to calculate the metrics in a unified way. Using common metrics enables the different groups and organizations to share a common view of the progress of the project. In this paper, we introduce this method of sharing metrics on model-based development data.

The remainder of this paper is organized as follows. In Section II, definition of the EAST-ADL Language, while metrics using path queries defined in Section III. Section IV presents the implementation aspects of XPath, while Section

[1]Synligare means “more visible” in Swedish.

V dissection and conclusion, and Section VI gives a vision for future work.

II. THE EAST-ADL LANGUAGE

EAST-ADL is a domain specific architecture description language specialized for describing automotive E/E systems. The language supports the use of different levels of abstraction with traceability between the levels. The logical structure of an architecture expressed in EAST-ADL is according to a structural component model where components are connected through ports.

EAST-ADL defines an exchange format in XML, called EAXML [12]. The schema of the EAXML is the most precise definition of the language, although the underlying meta-model is defined in UML. The mapping between the meta-model and the XML schema is according to patterns defined in the AUTOSAR community. According to these patterns the schema becomes a reflection of the meta-model, and the schema will only include elements according to the meta-model.

Note that the principles behind the EAXML and ARXML (AUTOSAR xml) schemas differ from the schema of the XMI format, used for the representation of UML models; XMI is based on the more generic MOF (Meta Object Facility) framework [13]. This means that the schema of XMI will not reflect the used meta-model, but rather the meta-meta-model according to MOF. A consequence of importance to the use of XPath is that the element structure of an EAXML file is a direct reflection of the corresponding EAST-ADL model.

III. METRICS DEFINITIONS USING PATH QUERIES

XPath 2.0 became a W3C recommendation 2007. XPath is a specialized query language that can express selection criteria of nodes of an XML document, typically from within an XML style sheet. The selection criteria include the path to traverse in the structure of the document, and additional tests and predicates that must be fulfilled for the selected nodes.

The way XPath is used is by 1) selecting the sets of nodes in the XML document that are relevant for the specific metrics, and 2) performing arithmetic operations on the quantities defined by the sets.

In this section, we present two types of metrics that we have specified with path queries and shared between object model tools. The metrics are inspired by the XPath query language for XML files. The first type of metrics calculates the progress of the development process using the product data. The second type of metrics calculated the complexity of the product components.

A. Progress metrics

One type of the metrics that we defined and shared between tools extracts the state of the project from the development data specified in different tools. The underlying specification of the tools is EAST-ADL, which enables us to create generic metrics and share them between tools. One such metric describes the completeness of the allocation of requirements.

The metrics value was originally expressed in the Synligare project as: "Progress of requirement allocation is measured as the fraction of requirements allocated to architectural elements"

The two sets of elements involved in this calculation are 1) the set of all requirements, and 2)the set of allocated requirements.

The first set can be expressed as the path expression (1) below, which is assumed to start from a "EA-PACKAGE" context node of the EAXML document. Definition for different elements of the XML representation of the meta-model such as EA-PACKAGE is available on EAST-ADL's language specification documentation [8].

Note that since the EA-PACKAGE structure in an EAXML document is an arbitrary packaging structure, it is suitable to exclude this part from the definition, and define the part on a case to case basis.

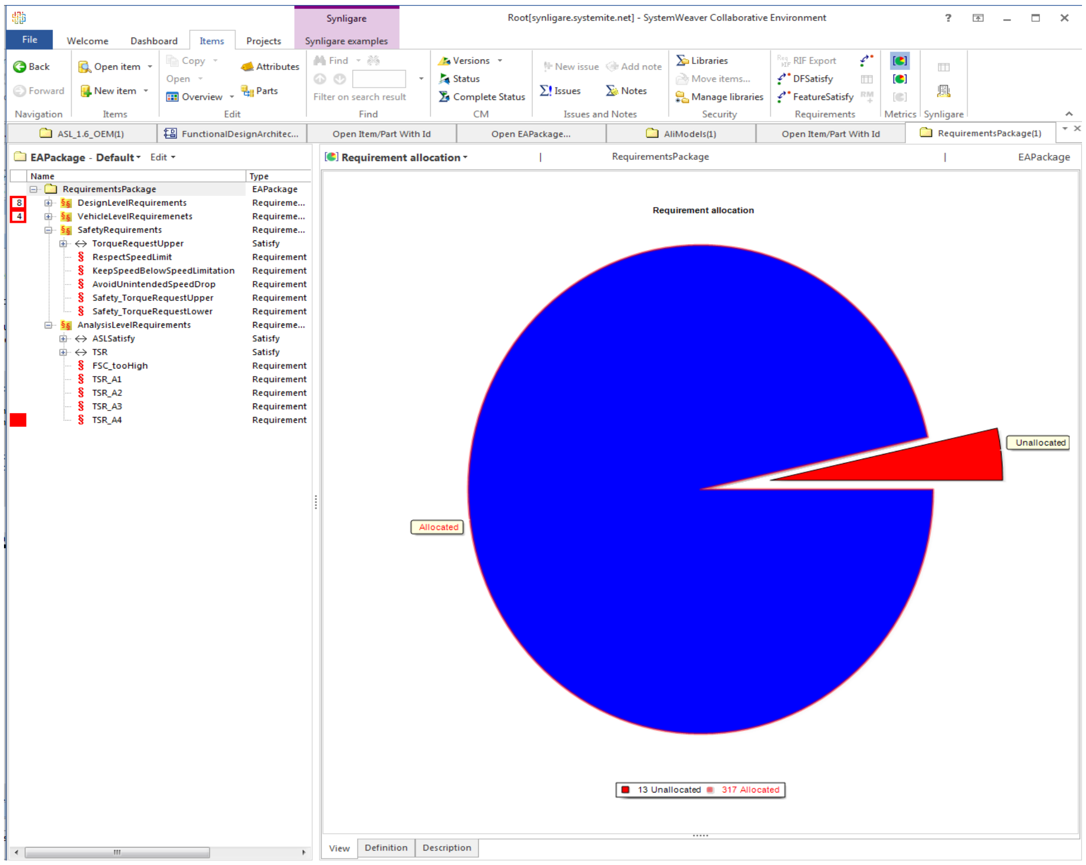
```
        /ELEMENTS/REQUIREMENTS-
  MODEL/REQUIREMENTS/REQUIREMENT        (1)
```

The set of allocated requirements is a subset of the set described above, with the additional constraint that the requirement must be included in a so called "Satisfy" relationship:

```
     /ELEMENTS/REQUIREMENTS-MODEL/OWNED-
        RELATIONSHIPS/SATISFY/SATISFIED-
  REQUIREMENT-REFS/SATISFIED-REQUIREMENT-
  REF                                   (2)
```

The set of unallocated requirements can be defined as the difference between the two sets, using the "except" operation:

```
        /ELEMENTS/REQUIREMENTS-
   MODEL/REQUIREMENTS/REQUIREMENT except
        /ELEMENTS/REQUIREMENTS-
  MODEL/REQUIREMENTS/REQUIREMENT        (3)
```



	Name	Type
	RequirementsPackage	EAPackage
8	DesignLevelRequirements	Requireme...
4	VehicleLevelRequiremenets	Requireme...
	SafetyRequirements	Requireme...
	TorqueRequestUpper	Satisfy
	RespectSpeedLimit	Requirement
	KeepSpeedBelowSpeedLimitation	Requirement
	AvoidUnintendedSpeedDrop	Requirement
	Safety_TorqueRequestUpper	Requirement
	Safety_TorqueRequestLower	Requirement
	AnalysisLevelRequirements	Requireme...
	ASLSatisfy	Satisfy
	TSR	Satisfy
	FSC_tooHigh	Requirement
	TSR_A1	Requirement
	TSR_A2	Requirement
	TSR_A3	Requirement
🟥	TSR_A4	Requirement

Figure 1 Completeness of allocated requirements

The fraction of the sets can be calculated using the XPath count function and div operator:

```
count(/ELEMENTS/REQUIREMENTS-MODEL/OWNED-
    RELATIONSHIPS/SATISFY/SATISFIED-
  REQUIREMENT-REFS/SATISFIED-REQUIREMENT-
REF) div count(/ELEMENTS/REQUIREMENTS-
MODEL/REQUIREMENTS/REQUIREMENT)          (4)
```

The real underlying need behind this metric is the need for traceability to the set of unallocated requirements. This traceability can be performed interactively using a pie chart representation of the set (3) in the SystemWeaver tool [14]. We see the evaluated system in the tree view to the left in Figure 1. The system is the reference system of the Synligare project, supplied by Volvo. The package "RequirementsPackage" has been selected, thereby selecting the context of the evaluation. The "Requirements allocation" view to the right displays a pie chart, where the two slices represent allocated requirements (in blue) and unallocated requirements (in red). By selecting the *Unallocated* slice, the set of model elements according to the XPath expression (3) become highlighted in the tree view.

B. *Complexity of component models*

Another type of metric that we investigated in this paper is the metrics concerning complexity of component models. One such complexity metric is cyclomaticcomplexity [5], calculated for a component model.

```
   count(/CONNECTORS/FUNCTION-CONNECTOR)-
count(/PARTS/DESIGN-FUNCTION-PROTOTYPE) +
 2                                       (5)
```

Another component complexity metric uses couplings between objects [6]

```
count(/CONNECTORS/FUNCTION-CONNECTOR) div
  count(/PARTS/DESIGN-FUNCTION-PROTOTYPE)
                                        (6)
```

IV. IMPLEMENTATION ASPECTS OF XPATH

In the Synligare project, support for metrics definitions expressed by the path query language was implemented in the SystemWeaver tool. SystemWeaver has a programmable meta-model and constitutes an internal database that can manage and integrate the content of multiple EAXML files. The constructs supported by the meta modeling framework in the tool supports the patterns used in EAST-ADL, like the type/prototype pattern. This means that the internal representation in SystemWeaver to a high degree conforms to the EAXML file format. A database like the one in SystemWeaver is not limited to managing the content corresponding to a single system, but can manage any number of systems, and content shared between the systems.

SystemWeaver supports dimensions of data that is not supported by EAST-ADL, like versioning and management of contexts that go beyond the scope of a single system. Such dimensions correspond to additional axes of the XPath expressions that cannot be derived from the specific meta-model.

A specific challenge is the way references are expressed according to EAST-ADL and AUTOSAR. Instead of common XML ID/IDREF to express references, EAST-ADL and AUTOSAR uses element paths of the XML file to reference elements, e.g.,"/DesignLevelElements/ FCN/GlobalBrakeController/BrakeTorqueFL".

References like the one described above are common in the AUTOSAR/EAST-ADL models and means that the XPath expressions cannot be evaluated against a DOM (Document Object Model). Instead, the XML file has to be parsed and transformed into a custom object model where references have been replaced by object links. SystemWeaver for example represents the references as bi-directional object links. During an import of an EAXML file into SystemWeaver all path strings are parsed and replaced with object links.

It can be assumed that any tool that supports EAST-ADL or AUTOSAR will have an efficient internal representation of such references. We have seen that a real life AUTOSAR XML file can be of the size of 10 Mbyte or more, including more than 100,000 elements. A corresponding EAST-ADL model would include even more aspects, and thereby more elements. This means that efficiency becomes a real concern, especially when the evaluation of metrics is done interactively, or when the complexity of XPath expressions are $O(n^2)$ or higher, for instance, when set operations are used.

V. Conclusion and discussion

In this paper, we presented a generic method to formalize metrics and share them between model-based data management tools. In the Synligare project, metrics originally expressed in natural language have been re-expressed in an XPath-like format and executed in different tools with identical results.

Being XML based, Xpath is intended for use with XML based representations. Since XPath is implementation independent it can work as a formal definition of the metrics, while also being executable.

Elwakil et al. [4] identified a number of advantages of using XQuery in metrics definitions for XMI based representations. These advantages have been found to hold also for XPath, being a subset of XQuery, for the case that data is represented in the more basic XML representations used for AUTOSAR or EAST-ADL:

- The XPath expressions can be expressed according to the meta-model of the used architecture language, meaning that the correctness of the expressions can be validated statically.
- The XPath language is standardized, technology independent, mature and wide spread.
- A tool implementation of the method may directly interpret and execute the XPath expressions. This makes it easy to try different metrics expressions in the tool implementation, without changing the tool itself.

In addition to these findings, the implementation of the support for XPath has taken benefit from the fact that XPath supports the selection of sets of elements, thus making it suitable for interactive analysis and traceabilitybetween the visualization of the metrics and the underlying data.

The solution has been demonstrated using industrial examples, with satisfactory performance.

There are some natural limitations and disadvantages of using XPath:

- The approach is likely feasible only for those cases where the language is expressed as XML; specifically, that the schema is a reflection of the used meta-model.
- Given the declarative characteristics of the language it is likely that not all types of metrics can be defined easily in the language. The use of XQuery as described in [3] has not been investigated for the type of representation used in the project, but may be an alternative for more complex types of metrics.

VI. Future Work

The evaluation of XPath for metrics definitions described in this paper was limited to the use cases of the Synligare project. It remains to evaluate the suitability of the approach for other types of metrics.

Acknowledgements

This work was supported by VINNOVA under the FFI Programme, Grant 2013-2196 Synligare.

ReFerences

[1] D. Goswami et al., “Challenges in automotive cyber-physical systems design.” In Embedded Computer Systems (SAMOS), International Conference on 2012,pp. 346-354. IEEE.

[2] R. Okuda, Y. Kajiwara, and K. Terashima, “A survey of technical trend of ADAS and autonomous driving.” In VLSI Technology, Systems and Application (VLSI-TSA), Proceedings of Technical Program-International Symposium on 2014, pp. 1-4. IEEE.

[3] M. Sharma, N. S. Gill, and S. Sikka,“Survey of object-oriented metrics: Focusing on validation and formal specification.” ACM SIGSOFT Software Engineering Notes, 2012, 37.6: 1-5.

[4] M. El-Wakil, A. El-Bastawisi, M. Riad, and A. Fahmy, “A novel approach to formalize and collect Object-Oriented Design-Metrics.” 9th International Conference on Empirical Assessment in Software Engineering, 2005.

[5] Y.Jiang, B. Cuki, T. Menzies, andN. 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. ACM.
[6] S. S.Rathore and A. Gupta, (2012, September). “Investigating object-oriented design metrics to predict fault-proneness of software modules.” In Software Engineering (CONSEG), CSI Sixth International Conference on2012, pp. 1-10. IEEE.
[7] Synligare Consortium: Synligare Project website. Available from: http://www.synligare.eu/ [retrieved: Dec, 2015].
[8] EAST-ADL Association: EAST-ADL web site. Available from: http://www.east-adl.info/ [retrieved: Dec, 2015].
[9] SysML. Available from: http://www.omgsysml.org/ [retrieved: Dec, 2015].
[10] AUTOSAR Development Partnership: AUTOSAR web site. Available from: http://www.autosar.org/ [retrieved: Dec, 2015].
[11] XPath. Available from: http://www.w3.org/TR/xpath/ [retrieved: Dec, 2015].
[12] EAXML specification. Available from: http://www.east-adl.info/2.1.12/eastadl_2-1-12.xsd [retrieved: Dec, 2015].
[13] MOF. Available from: http://www.omg.org/mof/ [retrieved: Dec, 2015].
[14] SystemWeaver. Available from: www.systemweaver.se [retrieved: Dec, 2015].

B
Appendix 2

Traceability Data in Early Development Phases as an Enabler for Decision Support
XP 2016 : International Workshop on Emerging Trends in DevOps and Infrastructure. Edinburgh, Scotland.

Traceability Data in Early Development Phases as an Enabler for Decision Support

Bashar Nassar
Chalmers University of Technology
Gothenburg, Sweden
bashar.h.nassar@gmail.com

Ali Shahrokni
Systemite AB
Gothenburg, Sweden
ali.shahrokni@systemite.se

Riccardo Scandariato
Chalmers and University of Gothenburg
Gothenburg, Sweden
riccardo.scandariato@cse.gu.se

ABSTRACT
Traceability information between requirements, architectural elements and the results of test cases can be used to unearth interesting relationships between the early phases of the software development process and the software faults in the end product. For instance, complex dependencies between features and software components could lead to an increased level of flaws in the code. Such patterns can be detected and visualized as early warnings to the relevant stakeholders (e.g., the architect or the project manager). Ultimately, a fully-fledged prediction model can be developed if enough historical information is available from previous software projects. In this paper we introduce a method for building a decision support system based on historic product data.

Keywords
Traceability link; Requirement; Architecture; Test; Early development phases; Fault prediction.

1. INTRODUCTION
The problem tackled by this work is the early prediction of software defects, in order to reduce maintenance costs and improve quality. The context of this paper is the use of information extracted from requirements- and architecture-level information to predict test failures[1], which represent a reliable proxy for software faults. Previous work investigated the relation between the requirements and architecture, and how the requirements drive the architecture. For example, Nuseibeh [1] developed the Twin Peaks model to describe the co-development of requirements and architecture. Previous work has also investigated the relation between architecture and test, and how properties of the architecture can be used to predict faults. For instance, the work of Rathore and Gupta [5] shows that design attributes like coupling, complexity and size are correlated to fault proneness. However, the state-of-the-art is rather limited with respect to the whole chain of this relation, i.e., starting from requirements, to architecture, and ending with test.

In our work, we use data from large industrial projects and leverage the traceability data between requirements, architecture and tests in order to develop a fault prediction model, which could be used to generate early warnings to the software designers. In this paper, we outline our vision and present some initial results developed in an industrial setting.

The identification of faults can be done in different places throughout the development process. However, the later discovery of flaws leads to large fixing costs. In this work, the objective is to anticipate the faults at the early stages of the development process. The novel contribution of this work is the use of the relationships between requirements and high-level design (i.e., traceability links) to visualize sensitive points in the software system and possibly predict faults. An additional merit of this work is the use of empirical evidence. Realistic data about end-to-end relationships between the early artifacts and the discovered faults are not readily available to the research community. To overcome this limitation, we have partnered with a company specializing on managing traceability across artifacts in large projects. The uniqueness of the work comes from the tooling environment that stores the data traceability information. The tool enables programmed traversal of traceability links, which in tarn enables continuous and dynamic data analysis in an actual industrial context

The paper is organized as follows. Section 2 discusses the related work. Section 3 presents the objectives of this research and how we can meet the objectives by using requirements and design metrics, together with traceability data in order to define a prediction model. Section 4 contextualizes our work and presents the concluding remarks.

2. RELATED WORK
Fault prediction has been intensively studied in academia and industry on different phases of development. A significant amount of the research has been done on the implementation level. Many researchers have tried to predict faults based on source code metrics such as size of code, code change history, source code complexity, complexity of the implementation processes, programing languages. One such study has investigated the effect of class size on fault-proneness [3]. Another study goes further to investigate the relation between the complexity of source code modification and the faults prediction [2].

Having early predictors (i.e., before the start of the testing or implementation phase) is useful and desirable as it can reduce the cost of software development, improve the quality, and increase the reliability of the system. In this respect, less research studies exists. For instance, some studies exist about predicting the system defects before the testing phase. A model using Object Oriented metrics was built to predict the faults during design phase. This model used neural network technique and Bayesian Regularization (BR) algorithm. The results of this study prove that the BR algorithm provides better accuracy than Levenberg-Marquardt (LM) and

[1] In our work, we use the failures of test cases as a proxy for actual software faults.

Back propagation (BPA) algorithms. In addition, the study finds that machine learning models are significantly used and provide superior results [6]. Moreover, other works exist about using design properties as predictors. For instance, Rathore and Gupta [5] investigated the relationship between the class design level object-oriented metrics with fault proneness of object-oriented software system. The results of their work showed that design attributes like coupling, complexity and size are correlated to fault proneness.

Over the last decade, researchers have studied on particular areas of the traceability problem and they tried to developing more complicated tooling, even they suggested many new research areas to be address [12]. Traceability has many uses it could be used to check if the requirements have been satisfied in advance development phase, like design and code. Also, it could help to assess and manage the impact of changing a system artifact, i.e., requirement, among many other artifacts[13]. Moreover, traceability has been used in fault based testing to generate test data to demonstrate the absence of a set of pre-specified faults [14].

3. DECISION SUPPORT FROM TRACEABILITY DATA

Our objective is to discover and highlight the critical relations between requirements and architectural design choices that could lead to software faults. To achieve our objective, we have analyzed three large projects from two companies. The raw data from these projects (primarily requirements documentation and architectural design) have been processed via an automated tool (which we built) in order to extract relevant metrics. In the investigated projects, the failure of the test cases is linked back to the appropriate requirements and design modules. Hence, the collected metrics also represent a starting point for the construction of a decision support system for the identification of software faults.

In the following sub-section, we briefly describe the raw data from the three projects, the collected metrics, the technical aspects of the metrics along with some challenges of extracting them in real-life projects, and the initial results for the decision support system.

3.1 Three Industrial Projects

In this work, we are using a data management platform called SystemWeaver, which is a holistic information management solution for systems engineering and software development [8]. The selection of this platform as a data model was based on its capability to manage the product development process on different level of abstraction, the full traceability between the product artifacts, and the support for testing. The raw data which we are using to investigate our research comes from two of the leading automotive companies in Sweden. They include both the software and hardware design for several mature and deployed products. The structure of the data is represented in Figure 1. The highest level of abstraction is the system level, which contains the system features and represents how the features interact with each other. For example, a vehicle includes the braking function, the cruise control function, etc. The second level is the analysis level, which contains an abstract description of the system architecture. The next level is the logical design, which is a more detailed description of the system design and can be viewed as a refinement of the level above. At the logical design level, the system design contains a set of connected components. Each component has a set of requirements, functions, connection ports and some other attributes. The bottom-most level in the figure is the hardware architecture, which represents the hardware-level design of a vehicle as a network of electronic control units (ECUs).

The traceability information takes different forms. First, it includes the relations (called T.W in the figure) between the product artifacts at the same abstraction level, for instance, the traceability links between a software component, its requirements, and test case validating the component's behavior. Second, SystemWeaver also maintains the relations (T.L in the figure) between the product artifacts on different abstraction levels. Finally, the traceability information could include the relations (T.V in the figure) between product versions.

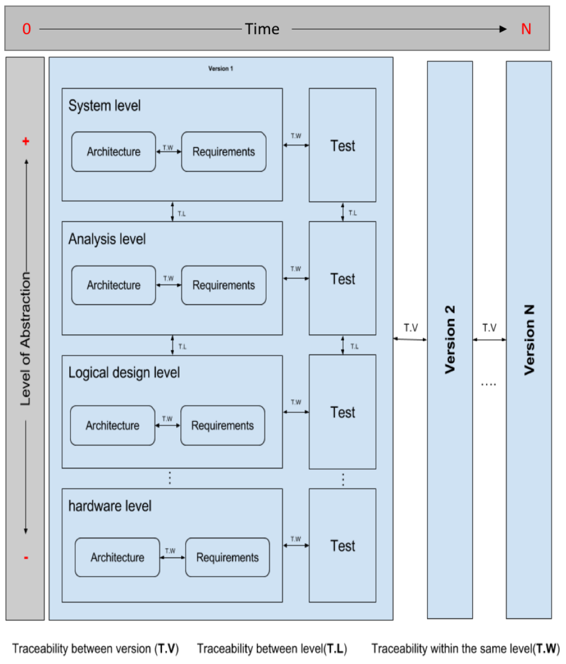


Figure 1. Traceability between system artifacts, abstraction level, and versioning.

3.2 Extraction of Useful Metrics

We have developed a module (called SmartTrace) that integrates with Systemite's SystemWeaver [9] and performs the automated extraction of several metrics out of the raw data mentioned in Section 3.1. This module calculates metrics that can be specified dynamically using path expressions, as discussed in [7] and Table 1 presents an example for some of the metrics that we had implemented in SmartTrace.

Metrics have been used intensively by the software community to measure the system attributes in different phase of the development process, in order to evaluate the quality of the system. As shown in the Table 1, the metrics that we extract for our decision support system include requirements, design, test, and traceability metrics.

The requirement metrics measure different aspects of the requirements like ambiguity, correctness, completeness, understandability, etc., [4]. The design metrics measure the design attributes, such as size, depth of inheritance, coupling, complexity, etc., [3]. The test metrics measure the testing attributes, i.e., number of test cases passed or failed, number of test cases under investigation and test case execution time. Finally, the traceability metrics measure the relations between different system artifacts, those artifacts are requirement, component, and test cases. By applying more metrics, the fault prediction module will be more powerful, which will increase the probability of finding the fault patterns. Table 1 presents some of the metrics that we had implemented in SmartTrace fault prediction module, some of them are conventional metrics, where other are more innovative.

Table 1. Metrics suite used by our decision support system.

Category	Metric	Description
Requirement	Requirement Unambiguous	It indicates the degree of the requirement understandability.
	Requirement Completeness	It indicates the degree of the requirement completeness
Design	Cyclomatic Complexity	It indicates the complexity of a system, by counting the number of the independent paths through a system.
	Coupling	It measures the degree of interdependence between software components.
Test	Number of Test Case	It measures the total number of performed test cases for a specific part of a system, regardless of the result of the test case.
	Pass/Fail Test Case	It measures the number of passed/failed test case for a specific part of a system.
Traceability	Component Responsibility	It measures the number of requirements implemented by a component .
	Requirement Uses	It measures the number of components implementing a requirement.

3.3 Metrics implementation and challenges

We implemented the metrics using the path expression language in SystemWeaver, which navigates through the elements and attributes of the product. The result of the executed query is a set of model elements according to the XPath expression. The language supports the basic model operators, like div, and, or, and so on[10].

The following is a simple example for one of the implemented metric, i.e., the component interaction metric (CIM) [11]. This metric measures the component interaction degree, by calculating the ratio between the component's incoming (I) and outgoing (O) interaction. The formula of the metric is CIM = I/O.

This metric can be calculated using the following XPath expression, which includes the count function and div operator:

```
/ELEMENTS/COMPONENT-MODEL/IN-
PORT.Count.Div(/ELEMENTS/COMPONENT-MODEL/OUT-
PORT.Count)
```

Using Xpath to define metrics has several advantages. For example, it allows us to share metrics between different parties, who have the same data model (meta model). This improves the communication between different tools or organization.

We had several challenges associated with extracting the metrics. For instance, the structure of the raw data could be different between products or companies, which required us to adapt our metrics definition to fit the studied data. Another challenge was to understand the data with very large size. Also, we faced some limitations in the XPath language, where not all types of metrics can be defined easily [7]. Last but not least, sometimes the raw data that we have is inconsistent or not complete, which required from us some manual activities to overcome the problem.

3.4 Towards a Decision Support System

The above-mentioned SmartTrace module is currently being extended into an information dashboard for decision support, which guides the engineers towards the identification of problematic components and requirements.

It is designed to mature continuously as the product development data in the organization grows. The data, often created by users, is used to generate insights on the patterns that are likely to result in failures. These insights are used to provide the user with decision support as shown in Figure 1.

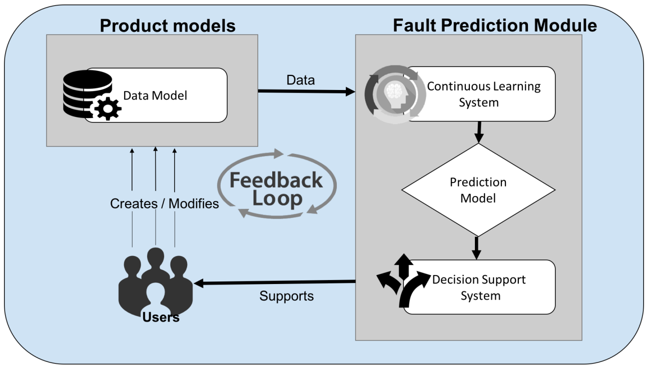


Figure 1. The SmartTrace fault prediction model using the traceability data in the early stage of development life cycle.

The users trigger the module by creating or modifying the product data. The traceability data between the product artifacts allows us to investigate the fault patterns, by applying statistical methods over the product data. We use the traceability between the product artifacts in addition to the isolated artifacts to retrieve these patterns. The first part of the module is a continuous learning system, which grows organically and continuously to update retrieved data from the product models, by adding the experience gained from new iteration to the old data. Over time the learning system will be more capable to provide the prediction module with necessary data to discover more potential faults.

Moving to the second part, which is the prediction model, this is responsible for predicting faults based on existing fault patterns, historical product data, and traceability data. The prediction module uses this data to find the faults or point to the places where the potential faults could be hidden. The last part of this module is the decision support system, which provides the users with real time decision support.

An essential part of this module deals with extracting the fault patterns. The patterns extraction process starts by defining and calculating metrics, these metrics measure a wide variety of product attributes, as mentioned before. The next step is to find the correlation between the calculated metrics and the related test cases, we use standard statistical tools, like non-parametric tests (e.g., Wilcoxon). A significant correlation with test failures means that the metrics can be used to identify the product elements related to the metrics (e.g., an architectural component) as a source of faults. The decision support system recognizes these occurrences and check the product based on the available patterns. Accordingly, the decision support system warns the developers about the potential risk and provides them with suitable suggestions (e.g., how to refactor the system to improve on the relevant metrics). For instance, if the number of dependencies shows a likelihood of introducing failures in one component, the decision support can suggest to split the component into two smaller components or to restructure the allocation of software components onto the electronic control units (ECUs).

Currently, we are looking for an automated way to discover those correlations between metrics and test cases. Later, we will expand this work to investigate the use of several metrics at once as failures could be due to factors related to multiple metrics.

4. CONCLUSION

This paper has reported on our vision concerning the use of rich traceability data to build a decision support system able to generate early warnings in case of problematic requirements, ill-designed architectural components, critical deployments and so on. The feasibility of our approach has been shown by investigating three large industrial projects and most of the necessary building blocks have been already developed. At this point, the key part of the proposed solution (the SmartTrace module) is being finalized. In future work, we will validate SmartTrace on a larger set of projects, possibly from different application domains, as so far we have focused on the automotive industry only.

From a research perspective, the decision support system we are developing is expected to significantly contribute to the early detection of requirements- and design-level flaws. The objective is to increase the quality and reliability of software by reducing the number of software faults and fail cases. From an industrial perspective, the approach is meant to reduce the risk of a new project, by applying the historical and accumulated knowledge from the projects previously developed in a company. In addition, it helps the designers to be more open to accept the customer-requested changes (as issues can be debunked by the decision support system), which will positively affect the customer satisfaction.

5. REFERENCES

[1] Nuseibeh, B., 2001. Weaving together requirements and architectures. *Computer, 34(3), pp.115-119.*

[2] Hassan, A.E., 2009, May. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering (pp. 78-88)*. IEEE Computer Society.

[3] Chen, J., Wang, H., Zhou, Y. and Bruda, S.D., 2011. Complexity metrics for component-based software systems. *International Journal of Digital Content Technology and its Applications, 5(3),* pp.235-244.

[4] Bokhari, M.U. and Siddiqui, S.T., 2011, March. Metrics for requirements engineering and automated requirements tools. In *Proceedings of the 5th National Conference.*

[5] Rathore, S.S. and Gupta, A., 2012, September. Investigating object-oriented design metrics to predict fault-proneness of software modules. In *Software Engineering (CONSEG), 2012 CSI Sixth International Conference on* (pp. 1-10). IEEE.

[6] Mahajan, R., Gupta, S.K. and Bedi, R.K., 2015. Design of Software Fault Prediction Model Using BR Technique. *Procedia Computer Science, 46,* pp.849-858.

[7] Söderberg, J., Shahrokni, A. and Nassar, B., 2016. Using XPath to Define Design Metrics. *International Conference on Performance, Safety and Robustness in Complex Systems and Applications (PESARO).* IARIA, 2016.

[8] Systemweaver.se, (2016). SystemWeaver - Managing product development information:: Home. [online] Available at: http://www.systemweaver.se [Accessed 14 Feb. 2016].

[9] Systemite.se, (2016). Systemite AB | Keeping complex systems consistent, correct and complete.. [online] Available at: http://www.systemite.se [Accessed 15 Feb. 2016].

[10] W3schools.com, (2016). XPath Tutorial. [online] Available at: http://www.w3schools.com/xsl/xpath_intro.asp [Accessed 16 Feb. 2016].

[11] Chen, J., Yeap, W.K. and Bruda, S.D., 2009, May. A review of component coupling metrics for component-based development. In *World Congress on Software Engineering* (pp. 65-69). IEEE.

[12] Cleland-Huang, J., Gotel, O.C., Huffman Hayes, J., Mäder, P. and Zisman, A., 2014, May. Software traceability: trends and future directions. *In Proceedings of the on Future of Software Engineering* (pp. 55-69). ACM.

[13] Cleland-Huang, J., Gotel, O. and Zisman, A., 2012. *Software and systems traceability* (Vol. 2, No. 3, pp. 7-8). Springer.

[14] Hayes, J.H., 2003, November. Building a requirement fault taxonomy: Experiences from a NASA verification and validation research project. *In Software Reliability Engineering, 2003. ISSRE 2003*. 14th International Symposium on (pp. 49-59). IEEE.