Západočeská univerzita v Plzni
Katedra informatiky a výpočetní techniky
Univerzitní 8
306 14 Plzeň

Assessing and Improving Quality of Safety Critical Systems

Odborná práce ke státní doktorské zkoušce

Štěpán Cais

Technická zpráva č. DCSE/TR-2015-02
Květen, 2015

Distribuce: veřejná

Technická zpráva č. DCSE/TR-2015-02
Květen, 2015

Assessing and Improving Quality of Safety Critical Systems

Štěpán Cais

Abstract

Safety critical systems are very important constituents of our lives. We use them daily and often without realizing it. For example, they can be found in control unit of a railway brake system, in aircraft's cockpit, inside missile guidance systems and many others. Because their failure can cause tremendous consequences, quality of these systems should be continuously assessed and improved.

Assessment of quality of safety critical systems should be transparent, objective and automated. By having quality assessed in these manners, the assessments' results are meaningful and can be also used for future comparison. One of the options for quality assessments are the software metrics. Their usage can produce fine data for system quality evaluation and also can point to weak parts of given system. Having the knowledge of the weak parts, the effort for improvement can be aimed directly at them. By improving specific part of the system, the overall quality can be enhanced heavily.

The topic of this work is the quality of safety critical software systems. For better understanding, the quality and safety critical systems will be discussed before the approach of assessing quality of safety critical system is described. Based on the assessment, method for quality improvement of safety critical system will be introduced and described.

Kopie zprávy jsou dostupné na:
http:/www.kiv.zcu.cz/cz/vyzkum/publikace/technicke-zpravy/

nebo na žádost poslanou na následující adresu:

Západočeská univerzita v Plzni
Katedra informatiky a výpočetní techniky
Univerzitní 8
306 14 Plzeň
Česká republika

1 Introduction

Nowadays, computers are almost everywhere. Besides personal computers which we are used to use on a daily basis, we can find them in many other devices. The endless list of devices having computer inside can start with smart tooth brushes measuring frequency of usage and can finish with very sophisticated safety critical systems orchestrating the train engines and brake systems. No matter which kind of computer is chosen, one thing is important overall – the quality of the inner software system. Although the quality is important in all the devices, it becomes crucial when the safety critical software systems are considered. The reason is evident – while failure in the software of smart tooth brush will at worst provide a meaningless statistic, the failure of safety critical software can endanger human lives and generate immense financial loss. The responsibility of safety critical systems indicates the importance of having quality of these systems assessed and also shows reasons why we should be trying to improve it constantly. The aim of this thesis is to bring meaningful survey into the area of assessing and improving quality of safety critical systems.

To improve the quality, firstly, we should be able to asses it. Usually, the assessment of quality of the safety critical systems is done by external authorities who are experts in the domain. The issue with assessing quality by experts lies on the subjectivity of the evaluation. For that reason, the requirements for assessing quality of safety critical system should be to have it transparent, objective and automated. The transparent, objective and automated assessment can produce meaningful results, which can be used in the future by other practitioners and also makes them comparable by each other. One of the means for assessing software quality is the software metrics. Usually, large number of software metrics can be measured automatically, generally with the high level of transparency and possible repeatability. The results produced by metrics measurement create data, which can point to weak parts of the system and help in quality assessment of the system. The assessment of software quality can be done by evaluation of the results of the metrics measurement.

The quality improvement can be done in several ways. One of possible ways for doing that is to select one or several important metrics from the set of metrics used for quality assessment and enhance the system according to them. By improving the system according to the important metrics, great improvement in software quality can be achieved. This approach can be very useful in the situations, where several weak parts of the system were identified as vulnerable or weak and where the budget of resources to improve it is limited. The concept of metrics importance can help to indicate on which part of the system we should focus firstly and to alleviate the overall process of improvement.

The thesis will cope with the topic as follows. Firstly, the general term quality will be revealed and perspectives on the software quality will be discussed. Software metrics as one of the major means in quality assessment will be introduced and their connection with software quality over the software quality models will be defined. Lately, the area of safety critical systems and methods used for their implementation will be covered. Afterwards, with theoretical background from previous parts, the quality evaluation of safety critical systems will be discussed with focus on evaluation of one particular real-world safety critical system. Later on, the topic of improvement of quality of safety critical systems will be explored. Subsequently, importance of metrics will be described and several methods will be discussed which usage can lead to system improvement. One of the methods will be

selected and deep analysis of its appliance to certain real-world safety critical system will be shown. Finally, the last section will provide conclusion about gathered results and several possible directions of future research will be discussed.

2 The Quality

To be able to understand the term of software quality, it is necessary to start with the definition of the quality itself. Quality is a term which anyone of us knows and use on everyday basis. We often use the word quality for expressing our satisfaction with some product or service. The quality has been here for long time and several styles of definition were coming up in previous decades. For example, so-called quality guru Feigenbaum [1] defines quality as:

“Quality is a customer determination, not an engineer’s determination, not a marketing determination, nor a general management determination. It is based on upon the customer’s actual experience with the product or service, measured against his or her requirements – stated or unstated, conscious or merely sensed, technically operational or entirely subjective – and always representing a moving target in a competitive market. Product and service quality can be defined as: The total composite product and service characteristics of marketing, engineering, manufacture and maintenance though which the product and service in use will meet the expectations of the customer”.

As we can see, Feigenbaum’s definition of quality is primary targeted on the customer and on the meeting of his needs. Another definition of quality provides Crosby [2]:

“The first erroneous assumption is that quality means goodness, or luxury or shininess. The word “quality” is often used to signify the relative worth of something in such phrases as “good quality”, “bad quality” and “quality of life” - which means different things to each and every person. As follows quality must be defined as “conformance to requirements” if we are to manage it. Consequently, the nonconformance detected is the absence of quality, quality problems become nonconformance problems, and quality becomes definable”.

As opposite to Feigenbaum’s definition of quality, Crosby is highlighting the importance of the conformance with requirements. Another definition of quality is provided from Stewhard in [3]:

“There are two common aspects of quality: One of them has to do with the consideration of the quality of a thing as an objective reality independent of the existence of man. The other has to do with what we think, feel or sense as a result of the objective reality. In other words, there is a subjective side of quality”.

The Stewhard’s definition can be considered as both highlighting the necessity of meeting customer needs and conformance to requirements. According to [4], these two approaches stand for the major philosophies of defining quality by quality gurus. Another quality definition can be found in ISO 9000 [5], which defines quality as follows:

"Degree to which a set of inherent characteristics fulfills requirements (…) where the requirement is lately defined as a need or expectation”

Similar definition is introduced by The American Society for Quality [6] which defines quality as:

“A combination of quantitative and qualitative perspectives for which each person has his or her own definition. In technical usage, quality can have two meanings: a. The characteristics of a product or service that bear on its ability to satisfy stated or implied needs; b. A product or service free of deficiencies".

Another definition is outlined by Six Sigma [7], which defines quality as “Number of defects per million opportunities.”

The wideness of quality definitions and its usage in many industries can also be seen on Wikipedia [8], where the quality is defined in 8 different articles (Quality business, Quality philosophy, Quality physics, Quality assurance, Quality factor, Energy quality, Logical quality and Vapor quality).

We can see from previous examples that the spectrum of possibilities how to define quality is quite broad and it is useful to put the quality definition into a context, otherwise the term can be understand in many ways. Because software domain is sufficiently extensive, it has its own quality definition. The topic of software quality will be discussed in the next section.

2.1 Software Quality

2.1.1 Introduction

In the previous section, the general term quality was discussed. In this section, we will focus more on software quality itself. As we could see in the previous part, there are plenty of definitions of quality. Similar situation is in the software domain, where several definitions of software quality can be found. In the next paragraphs, the most known definitions are presented.

2.1.2 Definition from Pressman

“Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software” [9].

2.1.3 Definition ISO 14598-1

“The totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs” [10].

2.1.4 IEEE Definition

“The degree to which a system, component, or process meets specified requirements. The degree to which a system, component, or process meets customer or user needs or expectations” [11].

We can see that all three definitions provide quite similar look on what the software quality is. The intersection between them is a conformance to “stated” requirements or needs, which can be found across them. The first two definitions also work with the term “implicit characteristics” or “implied needs” that can be seen as a prolonged definition of “stated” requirements from the view of inner software content. The last definition puts on place the term “meets customer or user needs”, which brings it closer to the Feigenbaum’s definition [1] of general quality than the other two definitions.

2.1.5 Software Quality Evaluation

The definition of the concept of software quality brings the possibility to evaluate it. The software quality evaluation can have many stakeholders – for example, software quality assurance team, customers, management etc. Any of these stakeholders has his or her particular intentions in the whole software developing process, but all of them have the same target – to have software with a good quality. Nevertheless, all three definitions stated above are still quite general for the purpose of software quality evaluation. For this reason, we need to define a method for evaluation of software quality. One method for evaluation of software quality is the software quality models.

2.2 Software Quality Models

2.2.1 Introduction

Models can be seen all around us. We use models for project managing (waterfall / agile), models for cost estimation (COCOMO) etc. The purpose of models usage is to lower the complexity of particular topic. One of the main reasons for using software quality models for quality evaluation is that they reduce the concept of quality into a few manageable parts. The overall software quality is described by division into quality attributes, which are connected with particular software characteristics. We can see the software quality models as a mean to evaluate the software quality by defining quality characteristics and metrics for their measurement.

At the moment, there are plenty of software quality models. For proper understanding of the issue of assessing the software quality by the software models, we will firstly need to define the software quality model itself.

The definition of software quality model in [12] is quite simple and understandable:

“A model with the objective to describe, assess and/or predict quality”.

We can also find another definition in [13]:

”Quality model is set of characteristics and the relationships between them, which provide the basis for specifying quality requirements and evaluating quality”.

For our purposes, we will use adjusted definition of software quality model from [14], which is more concrete:

“Software Quality Model is a set of factors, criteria and metrics (characteristics) and the relationship between them. These relations provide the basis for specifying quality requirements and evaluating quality”.

The term software metrics is used here but we will define it later. With the term software quality model defined, we can describe some of the most known quality models for obtaining better understanding of how the software quality can be evaluated.

2.2.2 McCall’s Quality Model

McCall’s quality model is one of the most known models and was used as a predecessor of today’s quality models. Importance of this model is highlighted by the fact, that the McCall’s model was used as a foundation for the ISO 9126 quality model [13]. The model was named according to the main author Jim McCall and was developed in collaboration of US Air-Force Electronic Systems, the Rome Air Development Center and General Electric in the 1970s (this model is also known as “General Eletrics Model of 1977”).

The lack of accepted definition for software quality and confusion in selection of quality goals for software were the main motivations for the model creation. The hierarchical three-level quality model was introduced – the idea of McCall’s Quality Model is that the quality factors should provide a complete software quality picture [15].The model is based on clear identification of relation between quality factors, criteria and metrics. The quality is identified via three perspectives:

1) Product operation – ability of the product to be quickly understood, operated and capable of providing results to the user.

2) Product revision – ability to repair changes, error correction and system adaptation.

3) Product transition – ability to adapt to new environments.

Each of these perspectives is connected with its own “quality factors”. So-called quality factors represent external view of the software from a customer or user perspective and describe different types of system characteristic. The model contains 11 different quality factors, each with relationship with a quality perspective. These quality factors are: correctness, reliability, efficiency, integrity, usability (for the product operation perspective), maintainability, flexibility, testability (product revision perspective), portability, reusability and interoperability (for product transition perspective).

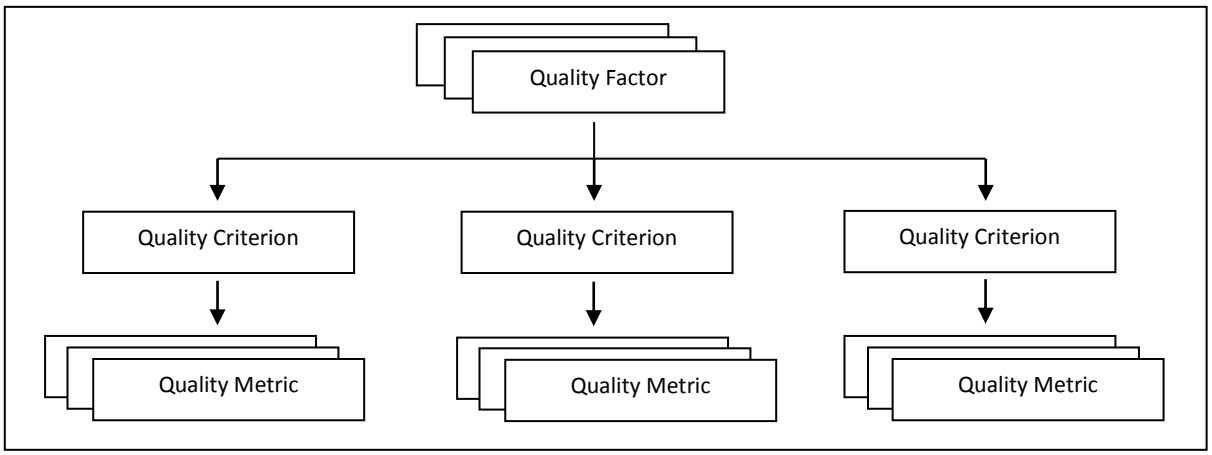

Figure 1. The hierarchy levels of McCall's model

The second level of the model is formed by “quality criterions”, which are attributes to one or more of the quality factors. The quality criterions represent an internal view of the software as seen by developer. The model contains 23 different quality criterions such as traceability, self-descriptiveness, consistency and so on.

The third level of the model is formed by “quality metrics”, which measure the particular attributes of software. The reason for having metrics in the model is to provide method for measurement. The model consists of 41 defined metrics. These metrics can be measured at various moments during development to give indication of software quality.

Application of the model is described by McCall in four steps:

1) Deduce quality factors based on the characteristics of the system.

2) Trade-off and prioritize the quality factors based on the needs of the customers / users.

3) Deduce related quality criteria and metrics using the framework.

4) Base specification, design, coding and testing on the deduced factors, criteria and metrics.

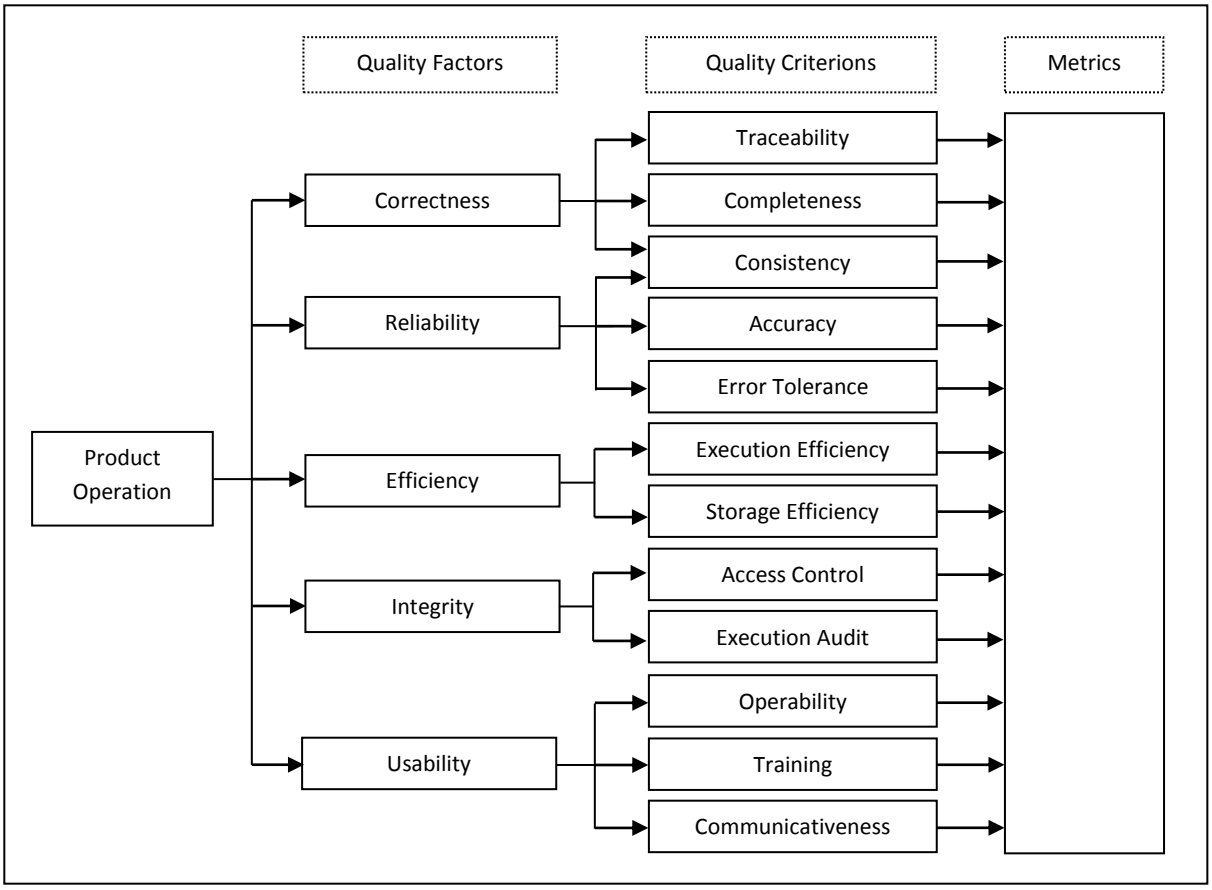

Figure 2. Product Operation linked with Quality Factors, Quality Criterions and Metrics

2.2.3 Boehm's Quality Model

Boehm's software quality model [16], [17] is another example of hierarchical model and has similar structure as the McCall's quality model. It is considered as another predecessor of today's quality models [18], mostly because broadens the McCall's quality model. Boehm introduced his model later then McCall and put more emphasis on the software maintainability. The maintainability in the Boehm's quality models is a quality factor which consists of three another factors – the testability, understandability and modifiability. This makes the maintainability dependent on testability. The model begins with the base point – general software utility – which represents the overall system quality. The general software utility is an aggregator of three factors:

1) As-is utility – how well can be the software product used as it is.

2) Portability – how easy it is to modify, understand and retest the software product.

3) Maintainability – how difficult it is to use the product when the environment changes.

These three factors are aggregators of another 6 factors (portability is also one of the factors, so the overall number of factors is seven). These quality factors are reliability, efficiency, human engineering (aggregates of As-is utility factor), testability, understandability and modifiability (aggregates of maintainability).

The factors are further divided into criterions, which represent lower level of software characteristic. In the Boehm's model, there are 12 quality characteristics: device independence, completeness,

accuracy, consistency, device efficiency, accessibility, communicativeness, structuredness, self-descriptiveness, conciseness, legibility and augmentability.

The criterions can be measured with 151 metrics whereas each metric is defined in form of a question, so they form a checklist. Boehm's model, in contrast with McCall's model, applies only to code, so it puts no emphasis on documentation, processes etc.

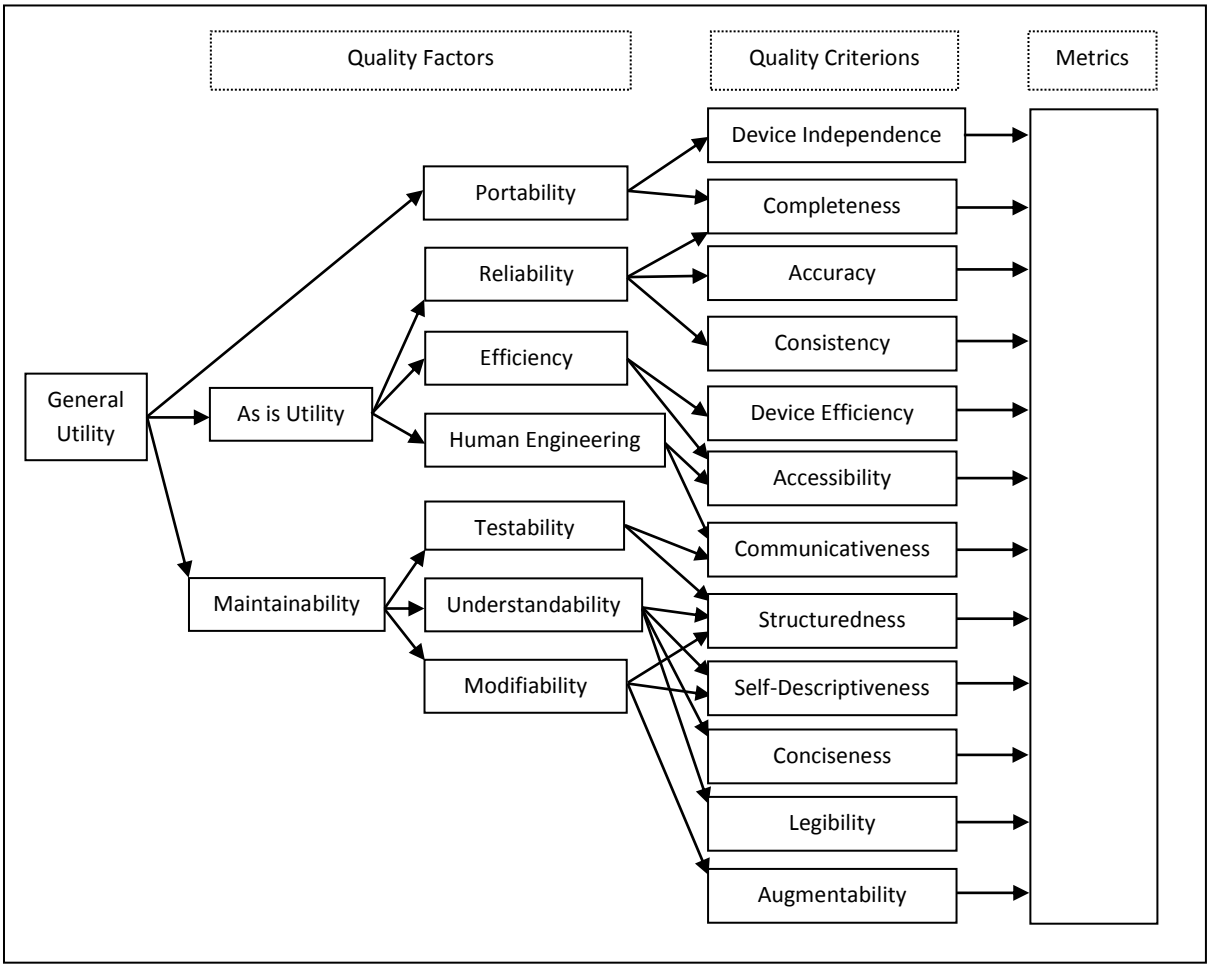

Figure 3. Boehm's Quality Model

2.2.4 ISO 9126 Quality Model

ISO family standards are well known, especially because of the ISO 9000 [5] series for process quality standards. The ISO 9126 was designed to bring standardization into the area of software quality. The standard has been evolving in the last decade quite heavily. Therefore, nowadays the whole standard consists of four parts. These parts are: ISO 9126-1: Quality Model [13], ISO 9126-2 External Metrics [19], ISO 9126-3 Internal Metrics [20] and finally, ISO 9126-4 Quality in Use [21]. Recently, the ISO 25000 [22] as the successor of ISO 9126 was introduced, but because it is still under development, it will not be considered in here.

The quality model itself is described in the document ISO 9126-1 [13]. The ISO distinguishes between three points of view regarding to the software quality. The quality can be seen from internal, external or so-called “in-use” perspective. The documents for ISO 9126-2 [19], ISO 9126-3 [20] and ISO 9126-4 [21] provide set of metrics for measurement quality from particular perspectives. The external

metrics provide quality evaluation of the system concerned as the system was running in a simulated environment with prefabricated data. Conversely, the metrics used for measuring the quality in use measure the effect of the software on users while the system is running. The metrics used for measuring the internal quality are focused on assessing the products quality characteristics, which do not have to be seen from outside world (for example, the maintainability).

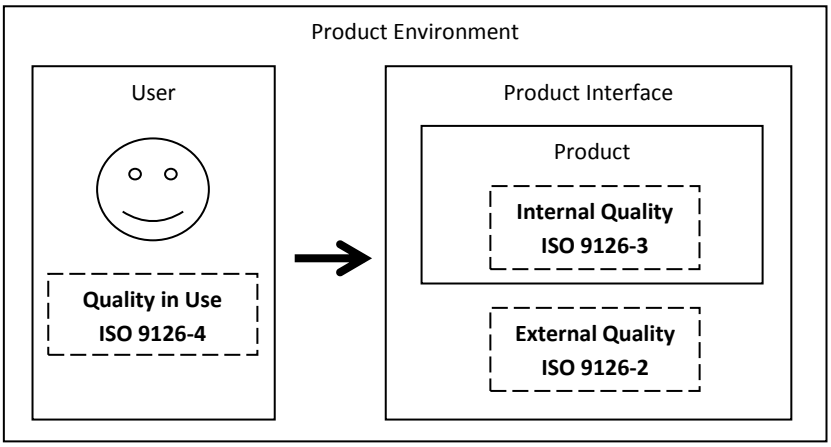

Figure 4. Linking parts of the ISO 9126 in Product Environment

The quality model of ISO 9126 is hierarchical and extends the quality models of McCall, Boehm etc. The model has three hierarchical levels, whereas the top level consists of six quality characteristics. The characteristics are functionality, reliability, usability, efficiency, maintainability and portability. In case of quality in use part, we need to add another four characteristics: effectiveness, productivity, safety and satisfaction. The characteristics for internal and external quality are further divided into 27 sub-characteristics. For example, the characteristic usability is connected with sub-characteristics understandability, learnability, operability, attractiveness, usability compliance. The sub-characteristics can be measured by the sets of metrics provided by the standard. The metrics proposed in the standard can be amended by metrics defined by user.

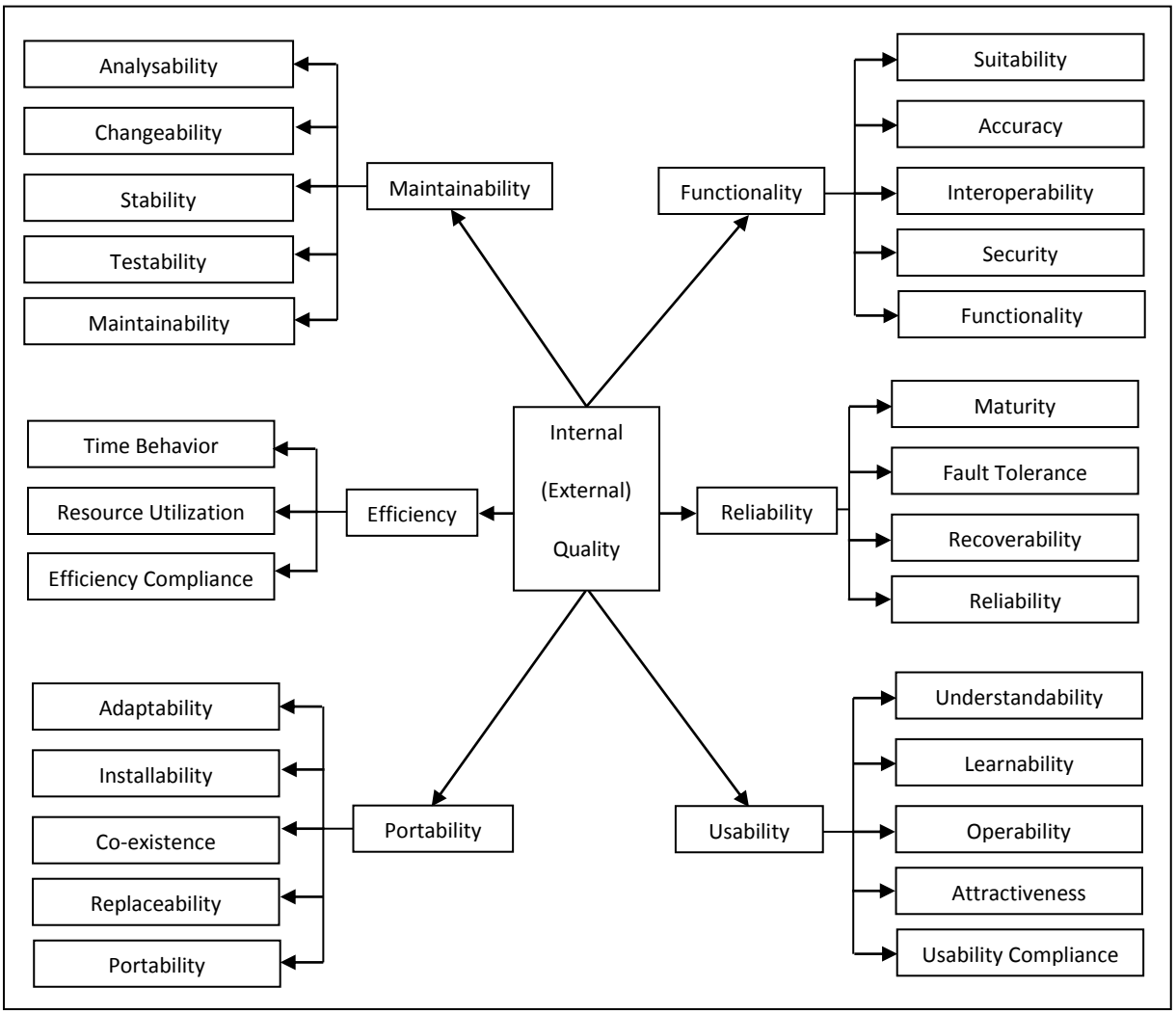

Figure 5. Characteristics and subcharacteristics of ISO 9126 Quality Model for Internal and External Quality

2.2.5 Problems with Quality Models

Even though the quality models represent a quantitative and systematic way of assessing software quality, we should still keep in mind that sometimes, their usage can be tricky. The issues of quality models have been discussed many times. For example, choosing a model for the purpose of software quality evaluation can be challenging [23] and organizations that want to use a quality model should be very familiar with their definitions. For example, in [24] we can see that even comparison articles of software quality models can give different points of view on the same models.

Furthermore, there is uncertainty about how much can a quality model help with software quality evaluation. The article [25] brings interesting findings about quality models applied on the same projects. The authors claim that different quality judgments can be gathered from evaluation of the same projects by usage of different quality models.

Nevertheless, we can still say that the quality models can be very useful when helping with evaluation of software quality and they can illuminate the way of achieving quality software. Usage of quality model also pushes the users to start thinking about quality in the beginning of the software life cycle and to control the quality all over its stages. At the moment, there are plenty of quality models and user is able to choose the most suitable for his needs. On the other side it is need to be

said that quality models should be chosen by experienced people in organizations with strong knowledge of their usage.

2.3 Software Metrics

Software metrics are used for measuring particular software sub-characteristic and are well-known means for software quality analysis [26]. We can see them as a mean for software analysis, which can be very helpful not just for software engineer and quality assurance analyst. The main aim of particular metric is to give the straight impression about evaluated software to whoever will be interested in this information.

We can sort software metrics to categories according to their usage. One of them is the program metrics used for measuring inner characteristics of an inspected system (i.e. number of lines of code, number of methods etc.). A subset of these metrics is a collection of object-oriented metrics and their goal is to measure the object-oriented software. Another example of different metrics can be the metrics relevant to project management (number of commits per day etc.).

There is a huge amount of object-oriented metrics and many of them were the subject of research and examination in the past [27]. Quite broad amount of metrics is provided by the quality models itself. It is common to group metrics into metrics suites according to their features or authors of the metrics. Most known metric suites are Chidambelr´s and Kemerer´s metrics suite [28], Li´s and Henry´s metrics suite [29], Bieman´s and Kang´s metric suite [30] and Hitz´s and Montazer´s metric suite [31]. Metrics from different suites can be used for quality evaluation with different software quality models.

There are many questions which user of metrics needs to be aware of before using them. First of them can be how to relate metrics to particular quality sub-characteristic. This important question is fortunately answered in some of the quality models. For example, the ISO 9126-3 [20] defines set of metrics, which can be used for evaluation of particular sub-characteristic but the model also contains information that “The user of these technical reports may modify the metrics defined, and/or may also use metrics not listed.” User can arbitrarily enhance the metrics used to measure the quality according to his intention; he can even remove some of the metrics or add another. We can see that the metrics´ set is not invariable even for ISO 9126-3 [20] and depends heavily on point of view of the person preparing the quality model, evaluator of the system, the inner software characteristics etc.

Another issue is connected with the wide range of metrics. At the moment, there are a lot of available metrics and it can be misleading to say which metric should be used for measuring particular software sub-characteristic. Answer to this question is not simple and depends on character of the software, purpose of metric usage etc. In [25] we can find that “Metrics are of little value by themselves unless there is empirical evidence that they are correlated with important external (quality) attributes”. This gives us partial answer to the question: the important software sub-characteristic needs to be identified and connected with proper metrics, before the metrics are measured. The metrics chosen for specific attribute have to be selected carefully and have to be correlated with the sub-characteristic. Otherwise the result of measurement can be wrongly interpreted. As there is a wide range of possible metrics that can be chosen for particular sub-characteristic, we need to consider the suitability of a metrics carefully.

If we choose our metrics' set and have all of the metrics connected to quality model software sub-characteristic, we can still question the metric's suitability for selected sub-characteristic. We can see the suitability as a metric's "weight" in connection with evaluated sub-characteristic. The problem of metrics' weight is still open. The problematic was partially discussed in [14], where the author connects metrics with quality attributes and defines their weight by categorizing them into groups with names as "related", "highly related", "inversely related" etc. Another approach can be found in [32] and [33], where the authors use special method for classifying metrics weight.

The next question is connected with software metrics' thresholds. If we measure particular system, we need to evaluate the gathered results and say whether the results are in defined ranges or too high / too low. At the moment, there are no standardized metrics' thresholds which could be used without exception to any software system. The problem in here is that the same software metric can give different results according to measured system character, used measurement tool, used software language etc. For example, metric Deep of Inheritance Tree (which, simply speaking, measures depth of a class within the inheritance hierarchy) will have different results for a class in Java (all classes derived from Object class by default) and in C++. The problematic situation with metrics thresholds, metrics usage etc. will be covered in the next sections.

3 Safety Critical Systems

3.1 Introduction

Nowadays, safety critical systems are widely used and we can encounter them in many domains. It is possible to observe their growing employment in the transportation industry, where they can ensure many functions. We can find them in airplanes, space shuttle´s cockpits or inside railway interlocking systems. Safety critical systems have to meet demanding criteria to get the permission for usage in such exacting environment because usually, they are responsible for lives and property. Consequently, safety critical systems should be designed and implemented with consideration of their responsibilities.

The main aim of this section is to introduce the area of safety critical systems and bring meaningful survey of lately used techniques in this domain. All described systems have been recently developed for railway or subway industry usage and various techniques were used for their implementation. Comparison of used techniques can firstly, examine their particular advantages and drawbacks and secondly, give advice for their future application.

3.2 Safety Critical Systems - Definition

Today´s society is very reliant on computer technology and we all are surrounded by computers almost everywhere. Safety critical systems constitute significant part of this technological surrounding and differ from classical computer systems in terms of responsibility. In [34], safety-critical system is a term which refers to systems, whose failure can endanger human life, cause economical loss or environmental damage. Responsibilities of these systems put huge demands on software reliability, because a minor error in safety-critical system software can produce failure of complete system. This is the reason, why these systems should be designed and implemented with extreme care.

3.3 Methods Used in Safety Critical Systems

There are many areas, where safety-critical systems are used. We can find them in cockpit computer inside aircrafts, in command centers of nuclear plants, inside modern weapons or in space shuttles. They can secure breaking control of a car or help to manage subway and railway traffic. Technologies of these systems are still developing, although many verified solutions are known from the past. One area where safety-critical systems are used over a long period is railway and subway industry. These systems can help to control the train speed and location, manage and secure communication between stations and trains or ensure interlocking. Many articles were written about safety-critical systems in railway industry owing to the European Rail Traffic Management System (ERTMS) [35], which is being implemented in many countries in Europe.

Safety-critical systems used in railway industry have to meet requirements defined in norms. One of them is a CENELEC (European Committee for Electrotechnical Standardization) norm whose content defines "process and technical requirements for the development of software for programmable electronic systems for use in railway control and protection applications" [36]. This document defines and recommends set of techniques for developing of safety critical software for railway industry. One of recommended technique is N-version programming (NVP). According to [37], NVP can be considered as a mean for achievement of system fault tolerance. The aim of fault tolerance is to make system resistant to faults. In case of an occurrence of a fault in a system which is fault tolerant,

the system should not crash, but continues working. In best cases, system still provides its full services without a loss of performance [37].

Another recommended technique in [36] is the Failure Assertion Programming (FAP). The FAP uses formal description of a problem to facilitate and control programmers work. Although FAP is not considered as a direct fault tolerant technique, FAP with combination of other techniques can be used instead of NVP [36]. According to [38], fault tolerant and complex systems should be created with respect to modular design, because it can bring needed reduction of size and complexity. Besides, modular design distributes global responsibility to particular modules, which enables designer to prevent him from creating system with a single point of failure. Modular design can really improve system only if interfaces of modules are defined excellently [38]. Greatly used technique for creating well-defined interface specification is right FAP. The FAP usage can indirectly affect the overall inner fault tolerance.

3.3.1 FAP Method Used in JAZZ

As was mentioned above, one of areas where safety is crucial is railway industry. The high demands for safety standards and responsibility for lives and freight pressure railway industry companies to produce the best edge cutting technologies all the time. Because of importance of overall railway safety, every device running on railways has to meet demanding requirements. Approving of usage of one particular device is often long-distance run.

The major Czech company aiming at railway signalization and transportation is AŽD Praha s.r.o. One of the long-term mission of AŽD was to design and create United Architecture of Safety Devices (in Czech it is JAZZ – Jednotná Architektura Zabezpečovacích Zařízení). The main reason for creating JAZZ was the idea of one common platform, which could be used as a base for majority of safety devices. The idea is shown in Figure 6. All the process of design, implementation, testing and finally, certification for commission could be done only once. As all the process is done just once, huge amount of time and money can be spared. Any other safety device made by AŽD could use attributes of this platform as a ground for its own functionality.

Simply speaking, JAZZ platform can be seen like a hard real-time system, which enables functioning of algorithms of particular safety device. The platform contains both hardware and software which together represents safety critical system. The software which is a part of the platform enables to run algorithms of safety devices. This part of platform can be seen as a real-time operational system, which ensures communication with hardware and guarantee means for facile run of particular algorithm. Although the platform enables spatial redundancy – it is possible to let operate two or more redundant JAZZ components – the main aim of this text is to describe the inner software approach, which was considered as a possible substitution of NVP.

The software part of the platform was made with strong emphasis on the implementation of FAP. The idea of FAP is in concept of client – provider relationship. Both clients and providers are in relationship with formal contract, which express rights and obligations between them. The relationship is represented by conditions. The main aim of FAP lies in checking these conditions. Conditions are checked before execution of group of commands (checking input condition), after of group of commands (checking output condition) or – in case of invariant – during the period when instance of a class is in observable state [39]. The observable time can be defined as a time after creating an instance of the class and before and after any call of public method of a class [39]. If any

of conditions is not fulfilled, program is stopped. If program is stopped because of violation of input condition, an error lies in the client. If there is a violation of output condition, an error lies in the provider. Because by implementing the FAP we add software check-points to the code, implementation of the FAP can be seen as a creation of “another” software version, which observes the correct usage of origin software. To use FAP in correct way, conditions should be defined during analysis and then described in software specification. The more the description is formal, the better because the aim should be to create formally verifiable conditions

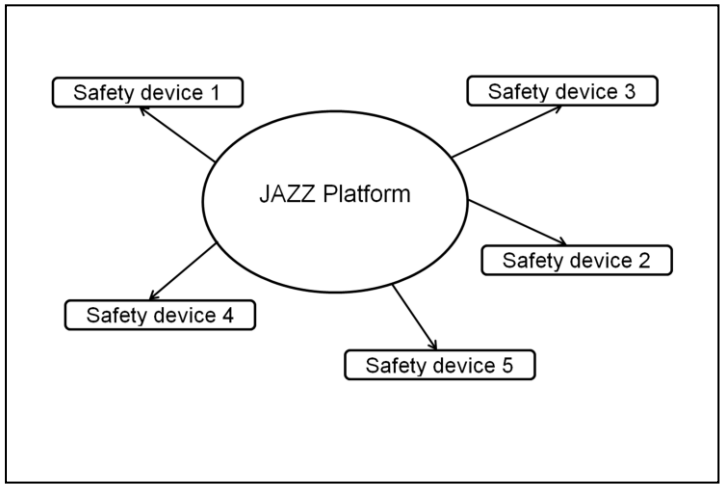

Figure 6. JAZZ platform

JAZZ source code is logically divided into two parts – descriptive part and operative part. Each part has its own responsibility. Operative part implements requirements from assignment and its main aim is to provide the major system function. We can imagine the first part as an implementation of real problem, for example, it can be the “code” of implementation of ADT stack. Descriptive part of the system enables checking whether system is working in accordance with given specification. The second part does not change the operational function, but describes the interfaces and possible states of objects. As far as the stack is concerned, the second part will ensure that stack does contain any element before the pop() function is called, or that stack has enough space to save another element before the push() function is called. We can say the descriptive part is the implementation of FAP.

Before the source code of the JAZZ was implemented, high quality specification was prepared. This specification was verified by software experts several times. Due to strict verification and N-times checking, final specification could be used as a source of conditions for the descriptive part of the system. After specification was created, implementation of JAZZ started. The JAZZ source code was written according to the specification. Because the specification contained very deep description of the system, the implementation of descriptive part was simply based on rewriting formal description from specification to code. The JAZZ was mainly programmed in C++ language. Therefore, macro techniques are used as a mean for implementation of the FAP.

There are many advantages of using FAP, a few of them will be described. One of the major advantages can be seen in comparison with NVP. In case of usage of NVP, basically N-identical programs are created from one common specification. With NVP, we rely on randomly discovered errors detected by comparing outputs from N-versions implementation. In case of FAP, we prefer

more active approach. The reached states of the system are controlled permanently and in case of violence of any specification rule the system is put out of operation. Because the rules are extracted from strictly defined and verified specification, we can rely on their correctness. Also the formal verification of program is less complicated than in NVP, because the specification is written in formal way.

Usage of FAP helps to reveal most of systematic errors and facilitate debugging and testing. Moreover, the usage of FAP brings systematic documentation directly into the code. By using FAP macros in JAZZ, the thoughts written in the code can be read in easier way. It helps another developer to get ideas from code faster and without much effort. On the other hand, any software programmer is limited by the strict and deep specification, so there is no much space left for his/her own creativity. It helps programmers to avoid needless mistakes and to concentrate on implementing what should be implemented. Writing specification into source code has also another benefit – the specification is kept up to date with the implementation.

Furthermore, in comparison with NVP, FAP is more resources sparing method. We need to implement only one version of software by only one team and no special supplements such as a communication protocol need to be created. Moreover, the specification already contains information about which conditions should be met, so the implementation is simplified. Although we need to invest more time into formation of the detailed and high-quality specification, in proportion to creating N-version software is FAP still sufficiently resource sparing technique. Generally, the technique of FAP leads to expense and work reduction.

The main drawbacks of FAP can be seen as a dependency on the specification, especially dependency on respecting the specification. For example, if a programmer would not keep the specification and omits any of defined conditions, system could act as being without problem although there may be an error. To reveal the omission of some conditions, review of source code has to be established and it brings an extra expense to overall cost. Furthermore, another disadvantage is the need of supporting process of development. The conditions have to be defined during the process of specification creating and programmers should be familiar with FAP before they start implementing the system. Another restriction can be seen in non-closable specification. Although the specification has to be created very precisely, in real word, there is a relatively high probability that some conditions will be overlooked or some conditions raise during implementation. Consequently, there is a chance there will be a need to add these conditions into specification. Owing to this, FAP can fit better for projects developed in agile development process instead of project with waterfall development process.

3.3.2 NVP Method Used in Railway Interlocking System

Nowadays, railway signalizing systems usually consist of three main parts – the Traffic Control Center (TCC), the Railway Field Components (RFC) and the Interlocking System (IS). Operator from TCC watches railway situations and gives needed orders such as route request or route cancelation to IS. RFC are the railway infrastructure, consisting of switches, track circuits etc. [40] Example of a signalizing system with TCC, RFC and Interlocking System represents Figure 7.

Interlocking systems are responsible for freight and passengers safety on railways and takes important role in decision making in railway signaling systems. Because of major role in safety and

decision making, interlocking systems have to be designed in an accordance of strict safety requirements.

Many articles about safety in railways, especially about safety in interlocking systems, were produced in recent years thankfully to The National Railway Signalization Project in Turkey [40], [41], [42], [43]. These studies describe safety application on railways and contain example of fault-tolerant technique. The fault-tolerant mean, NVP, is used in this project to ensure safety of the interlocking system.

In [40], [41], [42], [43], the IS is described of two parts – the Interlocking Modules (IM) and the Communication and Decision Making Unit (CDMU). The CDMU has many function, most important of them are communication between TCC, RFC, error logging and voting [40]. The CDMU works in cycles. In every cycle, data are read from TCC and RFC and then distributed to the IM. In the end of the cycle, data is collected from IM to CDMU and CDMU will process it. As a voter, CDMU needs to compare outputs from the IM and to choose the final result, which will affects the behavior of railway. The voting strategy of IS can be implemented in many ways, one of them is presented in [43]. If voter cannot decide about next step, the interlocking system comes to safe state. The safe state is also reached in particular situation, when IM do not make complete agreement. As an example, this situation may occur when at least one module did not approve the route acceptance [40].

Interlocking Modules process information for CDMU. Every module has its function implemented inside the Programmable Logic Controller (PLC). To implement the function of module, modeling techniques are used. To make every module independent to the others, all of them should be modeled by another team on another place and by using another modeling tools and techniques [42]. Firstly, behavior of the module is modeled by Petri Nets and then converted into fail-safe PLC code. This procedure is used because railway software needs to meet high software requirements given by CENELEC. According to [36], one of HR (High Requirement) is to use semi-formal methods for software, which needs to provide SIL 3 or SIL 4 (Safety Integrity Level) [44]. As Petri Nets are considered as a semi-formal method, they can be used for this modeling purpose.

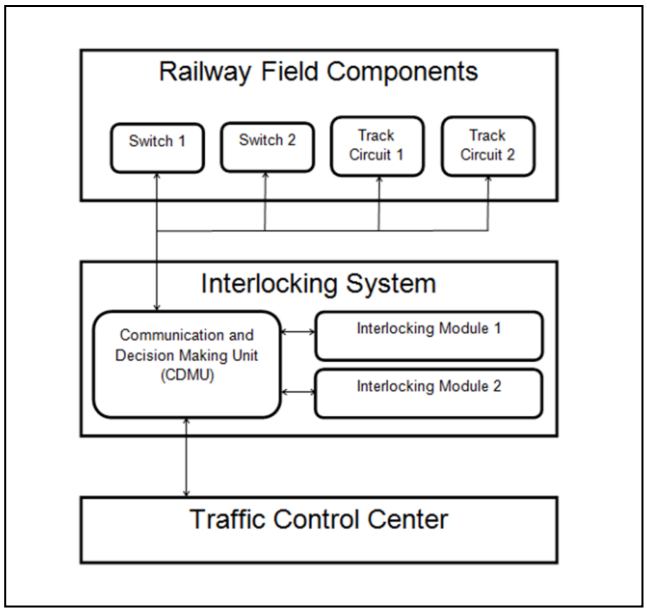

Figure 7. Signalizing system example

As a continuation of works [40], [41], [42] and [43] could be seen articles [45] and [46]. In [44], new voting strategy in diverse programming is introduced to improve the interlocking system. Ideas arising from previous works are followed in [46], where comparison between parallel and serial architecture of NVP is explored. The theme of the last article are the possibilities of implementation of NVP and their comparison. One possibility is a parallel architecture, where every module contains its own implementation of "decision" algorithm and modules are connected to the voter. Main issue of this principle is synchronization between modules and the voter because of communication latencies. On the other hand, there is a possibility to implement all "interlocking" algorithms in one module. Because of simplified communication (all algorithms are running on one PLC), the communication loads are smaller. Testing of these two approaches showed that the serial architecture has more advantages then parallel [46].

Although the algorithms of the PLC modules are modeled instead of programmed, we can still understand this example as NVP, because the principal idea (N-versions) is fulfilled. Main drawbacks of implementing this type of fault-tolerant technique are mainly known and stems from attributes of NVP. First difficulty is the cost of implementing this technique. To implement N-versions we need to have N-independent teams which would develop N-independent software versions. To keep the independence between teams, we need to guarantee independent team communication protocol. And to keep independence of software, we need to have both clear specification and different programming languages and tools that should be used. All this mentioned needs make the cost of implementing this technique higher.

Secondly, voter (in this case CDMU) is another problematic part of NVP. Without fault-tolerant voter, examination of program may crash without problems on the side of an independently implemented version. Moreover, communication between the voter and individual versions has to be considered, because faults in communication could lead to improper voter decisions. The voter is also responsible for switching system into the safe state. This state should be well defined and proper analysis of switching to safe state has to be made because without well implemented safe state switching, the system could spend too much time in the safe state, which could lead to overall inoperative behavior. On the contrary, if the system does not switch into safe state in a right time, an emergency accident can occur. Because of all these issues, voter is usually described as a critical part of N-version fault tolerant technique [47] and special emphasis should be deliberated before its implementation.

In previous text, we should notice the N versions are firstly modeled by semi-formal methods (Petri Nets) and then converted into PLC programming language by special tool - SILworX (www.hima.com). This method has its dark and bright sides. The main drawback is that this kind of implementing decreases the fault-tolerant technique because only one tool is used for both modeling and PLC code generation. In addition, the tool is used for software creation for all the versions. Although the tool is determined to meet high safety standards (SIL 4 certification), to create N independent versions, N different tools should be used for modeling as well as for code generation. On the other hand, there are advantages of this method. The usage of modeling and code generation tool facilitates the overall process of N-version software creation. To implement the software, we do not need to hire high-cost programmer, instead domain expert with Petri Nets experience could design the algorithms and then we can just let the code be generated. Another positive point is the possibility of formal

verification of designed algorithms. On the grounds of the usage of semi-formal methods, formal verification can prove the correctness of modeled algorithm.

3.3.3 Adjusted NVP Method Used in Train Control

In article the [48], a fault tolerant technique based on NVP has been explored. The technique has been studied for a usage in a fault tolerant system, which can be used in subway train controllers. The technique is connected to the method of controlling trains in subway – the Communications Based Train Control (CBTC).

The CBTC is based on constant radio communication between trains and the central system. The central system collects information about all trains in subway and evaluates gained data. In CBTC, every train communicates periodically with central system by sending information about its position and speed. Transferred information is used for evaluation of train movement and if the information signalizes violation of speed rules or possibility of danger, the central system issues a command to given train. The onboard controller responsible for providing information about the train state guarantees the communication with the central system. Both the central system and the onboard controller need to be fault-tolerant and safe, because fault of one of this system can emerge into catastrophic situation. For the central system, standard NVP is used. In article [48] is described the technique of improved NVP for the onboard controller.

The principle of altered NVP used in [48] lies in decomposition of the algorithm, which would be implemented in N-versions in standard NVP. The idea is: the more complex algorithm, the more errors established in implementation. The aim is to simplify the algorithm by division it to pieces. The example given in [48] shows following. Instead of having one complex algorithm responsible for speed evaluation, door opening and communication, at least five particular algorithms are created.

First algorithm is more robust than the others and performs control of the system. Although this algorithm provides control of the system, it is not performing any safety critical operations, but manages communication and provides information for user interface. The next two simple algorithms are responsible only for safety – according to [48], they are responsible for securing speed safety and managing right door opening. These two algorithms are implemented in style of NVP – any of these algorithms is implemented by N-teams in N-versions. The last two algorithms contain voting mechanism, first one for voting about speed setting and the second one for voting about door opening. The adjusted NVP idea is shown in Figure 8.

The modified algorithm also gives the ability to the safety algorithms to stop the train, even if only one of safety algorithms signalizes danger. This option allows to any safety algorithm which would discover unsafe conditions or which could leave to train crashing or derailing to stop the train. The impact of this attribute can be described as following. If only one of safety algorithms is correct, overall safety will be still guaranteed. On the other hand, if the algorithm is incorrect, it could cause superfluous suspension of a train.

Main benefit of this decomposition of NVP lies in simplification of critical parts of the system. Unlike classical NVP, this approach reduces overall complexity of the N-implemented algorithm by dividing the component into more parts. Critical parts responsible for system safety are minimalized to only implement its safety functions, nothing more. Owing to this simplification, formal verification of these algorithms and the proof of correctness of these algorithms are easier. Also the voting

algorithms are simplified, because every voting algorithm votes just about one type of inputs. This enables to reduce complexity of voters and facilitate their implementation in fault tolerant manner. In addition, correctness of the robust algorithm does not need to be proven, because this algorithm would never perform safety-critical operations. Other kinds of advantages come from the principle of NVP, which were described in previous part.

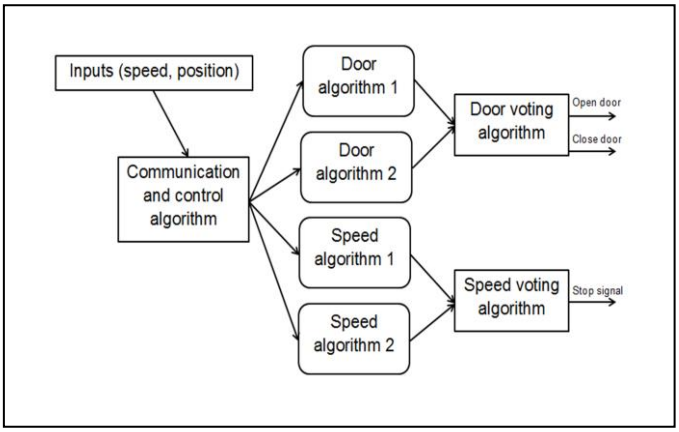

Figure 8. Adjusted NVP algorithm

Drawbacks of this approach can be divided into two groups. In the first group, there are drawbacks caused by the simplification of the original NVP algorithm. Although the division brings simplification of safety-critical algorithms, it enlarges the intensity of overall communication in the system. It is caused by higher number of voting algorithms and implementation of more particular safety critical algorithms. More transferred data puts emphasis to the communication system, which need to deliver more information reliably. Another drawback can be seen in the division idea itself. We can imagine systems, which would not be possible to split up (automatic subway train, where the speed is controlled by external device). In these kinds of systems, this idea need not be applicable. Nevertheless, as [48] demonstrates, we can imagine that division is reasonable. The second group of drawbacks is the list of drawbacks mentioned in previous sections and caused by the NVP itself. Unfortunately, this adjustment of NVP does not improve all issues of NVP (high costs, need of more teams for implementation, exact specification etc.).

3.3.4 Methods Comparison

Three methods used in railway and subway industry were described. Every method was firstly described and then its drawbacks and advantages were analyzed. First studied method was based on FAP technique and was implemented in JAZZ project in company AŽD Praha s.r.o. This method puts emphasis on strict software specification and definition of conditions, which need to be fulfilled to let system work correctly. Main advantages of this method lie in simplification of formal verification, linking between specification and the source code and better code transparency. On the other hand, it is needed to create strict and relatively deep specification. Moreover, only experienced and dependable programmer should implement this method, because omitting of any condition can cause unpredictable system behavior. To be sure about the conditions in the source code, reviews should be done.

The second method was based on NVP and was developed for railway interlocking system. In this case, the NVP is realized by modeling N-versions of software by a special tool, which enables converting models into code. This modeling approach brings the possibility of formal verification of

correctness of modeled algorithms, because models are originated by Petri Nets. Also the costs are lower than in classic NVP implementation, because we do not need to pay expensive software engineers for implementing the system. On the other hand, only one modeling software tool for modeling and generating all versions is used in this approach. In NVP, different tools should be used for modeling and generating. In addition, even though the used software meets high safety standards, we should not rely only on one tool. The third described method was an alteration of NVP used in the subway safety control system. The adjustment is based on dividing the N-version algorithm into more parts, so the safety critical operations are individually implemented, any of them in N-versions. This method facilitates the formal verification of algorithms and simplifies the voters. These advantages contrast with higher communication demands and sometimes exacting process of dividing one algorithm into individual parts.

Every of method analyzed has advantages and disadvantages and it is not possible to say generally if one is better than the other. Suitability of all methods depends on the purpose of particular system, as well on accessible means and time. The cheapest method can be considered as the one based on FAP, even though it is also connected with additional costs. The other two methods rely on NVP approach, where the need of N-teams brings considerably extra costs. The third example could be considered as a little bit expensive because we need to invest more into division of the algorithm. In the point of view of time demands, implementation of NVP from scratch can take longer time than the method with FAP. Nevertheless, with already defined communication protocol and other needs, NVP could be implemented in a shorter time than the FAP method. All three methods emphasize the simplification of formal verification. The best way for formal verification seems to be the second method because the system is defined by semi-formal mean, Petri Nets.

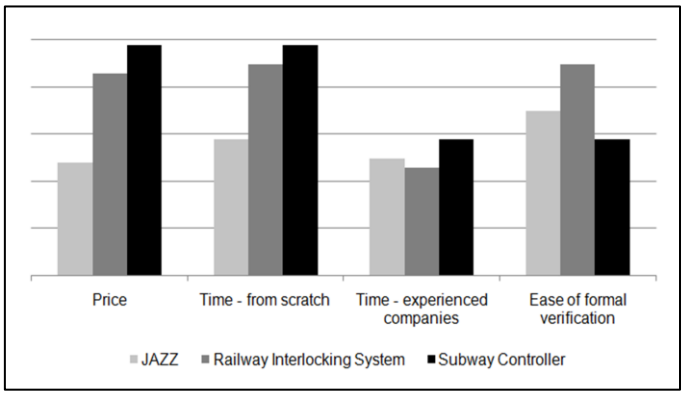

Figure 9. Comparison between methods

To conclude, ideal system should be created with respect to all three described methods. It should be designed as a NVP and the algorithms should be divided in individual parts as it was done in the third method. During the analysis of the system, strict specification ought to be delivered and used as a base for modeling the system. Different modeling tools should be used for modeling the system and subsequently for generating the source code. Conditions checking should be included into the source code, so the code remains easily readable and resistant to possible future bug seeding enhancements. Unfortunately, overall development of this ideal system would take a lot of time and money.

4 Evaluation of Quality of Safety Critical Systems

In the first section, the quality of software was discussed. Several quality models were introduced and their purpose was described. Also, the mean for measuring software quality, the software metrics, was revealed with focus on open questions in the area. In previous section, the topic of safety critical systems, their definition and methods used in them was described.

The topic of this section is the evaluation of quality of safety critical systems. Firstly, the software metrics relevant for safety critical software will be discussed. After that, tools enabling measurement of software metrics will be described and examined for the ability to measure the metrics relevant for safety critical software. Later on, sample project will be analyzed by the tools and results will be briefly discussed. Afterwards, thresholds for selected metrics for measuring safety critical software will determined. Finally, the quality of particular safety critical system will be evaluated based on the established thresholds.

4.1 Software Metrics Relevant for Safety Critical Systems

As it was discussed in previous sections, there are various software metrics and it is not unambiguous which one should we choose for system measurement. The quality models can give us some clues, but the metrics' sets are not exhaustive and more than focusing on measuring particular system, they provide general overview of possible measurements. This situation gets even more complicated when a safety critical system is the targeted application. Choosing right metrics for measuring safety critical software is delicate because usually, this kind of software is responsible for human lives, device safety etc.

Unfortunately, at the moment, there is no well-known and widely accepted exhaustive standardized set of software metrics aimed to measure safety critical systems. The situation is partially caused by wide spectrum of safety critical systems and norms recommended for their development. Consequently, companies developing safety critical systems are made to use metrics described in common software quality models, adopt metrics used in previously developed safety critical systems, develop its own metrics or just to take what is most suitable for them.

Two sets will be used to select metrics relevant for safety critical systems in here. First set is a metrics' set defined by Software Assurance Technology Center (SATC) at NASA Goddard Space Flight Center in their metrics' framework [49]. This set is widely cited and stands as a model for the second set. The second set of software metrics for safety critical software was developed by major Czech company involved in railway industry and was used for evaluation of quality of real-world safety-critical software.

The NASA framework contains description of 9 software metrics, which can be used for system evaluation and also as an example of metrics usage. The description of the metrics is given and in the end, the measured values of the metrics are discussed. The set contains 9 software metrics whereas 6 of them are considered as object-oriented and thee as traditional.

N.	Metric name	NASA classification	Construction
1	Cyclomatic Complexity	Traditional	Method
2	Lines Of Code	Traditional	Method
3	Comments Percentage	Traditional	Method
4	Weighted Method per Class	Object-oriented	Method

5	Response for Class	Object-oriented	Class
6	Lack of Cohesion Methods	Object-oriented	Class
7	Coupling Between Objects	Object-oriented	Class
8	Depth of Inheritance Tree	Object-oriented	Class
9	Number of Children	Object-oriented	Class

Table 1. Metrics' set from the NASA framework [49]

The real-world metrics' set contains 9 metrics that are almost identical, no classification was provided. Most of the metrics in the set measure class / object properties, just two of them measure methods.

N.	Metric name	Construction
1	Cyclomatic Complexity	Method
2	Source Lines Of Code	Method
3	Source Lines Of Code	Class
4	Number of Methods	Class
5	Response for Class	Class
6	Lack of Cohesion Methods	Class
7	Coupling Between Objects	Class
8	Depth of Inheritance Tree	Class
9	Number of Children	Class

Table 2. Metrics' set from the real-word safety critical system

To select metrics relevant for safety critical software, two described sets will be used. The relevant metric set arises by intersection of the mentioned sets. Some of the metrics described in both sets measure the same software attribute but have different names. In that case, the name used in the real-world metric set will be used. The final set of metrics relevant for safety critical system is shown in the following table. The short description for each metric is taken from the source set. If the metric is present in both sets, then the real-world description is used. The description in this table serves just as short characterization without focusing on deeper implementation details.

Metric name	Abbreviation	Numbers in sets	Short description
Cyclomatic Complexity	v(g)	1, 1	Possible paths through the program methods during execution
Source Lines Of Code (Method)	SLOC1	2, 2	Number of lines of code except for the lines containing comment only of method
Source Lines Of Code (Class)	SLOC2	x, 3	Number of lines of code except for the lines containing comment only of class
Number of Methods	NOM	4, 4	Number of all methods of given class
Response for Class	RFC	5, 5	Number of all methods of given class, the inherited included
Lack of Cohesion Methods	LCOM	6, 6	Percentage of all inner methods of the class, which use particular instance variables
Coupling Between Objects	CBO	7, 7	Number of other classes coupled to the analyzed one
Depth of Inheritance Tree	DIT	8, 8	Depth of a class within the inheritance hierarchy

Number of Children	NOC	9, 9	Number of immediate subclasses of given class
Comments Percentage	CP	3, x	Number of comments divided by the total lines of code

Table 3. Metrics relevant for safety critical software

The Table 3 contains 10 metrics which were selected as relevant for measuring safety critical software. Eight of the metrics were inside both sets, one metric, SLOC2, was added from the real-world safety critical software set and also one metric, CP, was added from the NASA metrics' set. All the metrics in the table measure the inner characteristics of the software, i.e. class / method.

4.2 Tools for Measuring Software Metrics

In previous section, set of metrics relevant for safety critical software was selected. In this section, some of the tools enabling automatic measurement will be briefly described and selected metrics' set will be mapped to measurement possibilities of these tools. The goal is to discover whether all the selected tools are able to measure set of metrics selected. Furthermore, a simple project will be analyzed with the tools and results gathered from the measurement will be discussed in short.

At the moment, there is a wide set of tools which enable measuring of software metrics. The tool can be a standalone application or it can be included in a framework, which can provide static analysis or similar functionality. Some of the tools available are open-source, some of them contain strict licensing so the distribution differs from tool to tool. The programming language which each tool supports, set of metrics possible to measure, the metrics' definitions etc. stand for difference too.

As there is a significant amount of tools providing measuring available, we will focus only on the tools providing metrics' measurement for C++ (which is common language for safety critical systems [50]). Furthermore, as there are several tools available even just for the C++ language, we will choose three of them: Understand, QA Framework and CppDepend. The first two were chosen because of author's experience with them; the third was selected because of several positive references.

4.2.1 Sample Project to Analyze

For metrics' measurement, an open-source project was selected. The project's name is Celero and can be found on GitHub repository [51]. The Celero is a benchmark tool providing easy interface to rate projects' performance. According to the author, the aim of the project is to "perform benchmarks on code in a way which is easy to reproduce, share, and compare among individual runs, developers, or projects" [51].

The source code is written in C++ and is well documented. Also the code is ready to be built in multiple environments and contain several examples of usage. We can consider Celero as a vivid (last commit April 2015) and popular (18 forks, 88 stars and 18 watches on April 2015). The Celero was chosen for our analysis because of all the mentioned positives.

4.2.2 The Understand Tool

Understand is a static analysis and metrics measurement tool from company Scitools [52]. The tool is multi-language and cross-platform and provides wide set of metrics to be measured. User can download the tool directly from the web pages and try it for 2 weeks for free.

The metrics' set contains around 100 metrics. Some of the metrics can be measured in most of the supported languages (for example, Average Lines of Code) and some of them are usable only for particular languages (for example, Number of PHP files). For C++, Understand contains about 70 metrics to measure.

For metrics' mapping, the C++ version of Understand, Understand C++, is used. The metrics for C++ are divided into four groups: project / file / class / method. As we can see in the Table 4, all selected metrics from the picked set were mapped into the Understand C++ API.

Metric's abbreviation	Name in Understand	Reference in the Understand metrics' API
v(g)	Strict Cyclomatic Complexity	CyclomaticStrict
SLOC1	Source Lines of Code	CountLineCode
SLOC2	Source Lines of Code *	CountLineCode *
NOM	Local Methods	CountDeclMethod
RFC	Methods	CountDeclMethodAll
LCOM	Lack of Cohesion in Methods	PercentLackOfCohesion
CBO	Coupling Between Objects	CountClassCoupled
DIT	Depth of Inheritance Tree	MaxInheritanceTree
NOC	Number of Children	CountClassDerived
CP	Comment to Code Ratio	RatioCommentToCode
* The metric can be used for measuring both classes and methods		

Table 4. Selected metrics mapped to Understand API

4.2.3 The QA Framework Tool

QA Framework is static analysis tool developed by the Programming Research Ltd. [53]. The framework is multi-language and cross-platform and allows user to operate with extensive set of code checks. Beside other things, part of the framework enables also the metrics' measurement.

The tool provides around 50 metrics to be measured. The metrics are divided according to their extent into project / file / class / method groups. The Table 5 contains mapping of the selected set of metrics into the QA Framework metrics API. All the metrics selected can be mapped into the QA Framework.

Metric's abbreviation	Name in QA Framework	Reference in the QA Framework metrics' API
v(g)	Cyclomatic complexity	STCYC
SLOC1	Number of code lines	STLIN
SLOC2	Total unpreprocessed code lines	STTPP
NOM	Number of methods declared in class	STNOM
RFC	Response for class	STRFC
LCOM	Lack of cohesion within class	STLCM
CBO	Coupling to other classes	STCBO
DIT	Deepest inheritance	STDIT
NOC	Number of immediate children	STNOC
CP	Comment to code ratio	STCDN

Table 5. Selected metrics mapped to QA Framework API

4.2.4 The CppDepend Tool

CppDepend is static analysis tool from company CoderGears [54] which provides several static checks for C and C++ code. The tool is distributed as standalone application only for Microsoft Windows platform and it can also be downloaded for evaluation.

The tool enables measurement of 80 metrics. The grouping in CppDepend is different from the previously described tools – the tool divides metrics into 7 groups and does not contain class / object group. Instead of that, the "type" group is provided. The Table 6 gives information about how metrics from the selected metrics' set can be mapped into CppDepend. We can see that all metrics but one, Response for Class, can be measured by the tool.

Metric's abbreviation	**Name in CppDepend**	**Abbreviation in the CppDepend metrics' API**
v(g)	Code Source Cyclomatic Complexity	CC
SLOC1	NbLinesOfCode	LOC
SLOC2	NbLinesOfCode	LOC
NOM	NbMethods	NbMethods
RFC	Methods	-
LCOM	Lack of Cohesion Of Methods	LCOM
CBO	Efferent Coupling at type level	Ce
DIT	Depth of Inheritance Tree	DIT
NOC	Number of Children	NOC
CP	PercentageComment	PercentageComment
* The metric can be used for measuring both type and method		

Table 6. Selected metrics mapped to CPPDepend API

4.2.5 Tools Results Differences

We can find in [55] that different tools can give different results when measuring the same metric on the same project. This information was also confirmed by the measurement of the sample project by the selected tools. Because it was not intended to provide deep analysis of this phenomenon, only some of the differences will be listed. Reader interested in this issue should have obtained enough information from previous sections to make his or her own measurement to investigate the differences more deeply.

	RFC [class Result]	v(g) [method celero::console::SetConsoleColor]	CP [class Experiment]	CBO [class Junit]
Understand C++	16	17	0.13	2
QA Framework	22	17	0.247	1
CppDepend	-	18	0	28

Table 7. Measurement's differences examples of the sample project

In Table 7, some of the measured differences are listed. The differences originated mostly due to different interpretation of particular metrics' measuring. As the C++ language is complex to analyze and there is no common standard with sufficient depth to determine how to measure metrics, the tools produced different results. The comparison of the metrics counting of these tools will not be covered in here because of the public unavailability of the documents / source codes which would have otherwise provided sufficiently deep insight into the metric counting. For example for the CP

metrics, we can imagine that the measuring is dependent on whether the comments of conditionally included files are also counted, which kind of comments the tool considers as comments, how the rounding is performed and so on.

4.3 Identifying Software Metrics Thresholds for Safety Critical System

In the previous part, software metrics relevant for safety critical system were selected. After that, three tools enabling measurement of these metrics were chosen. All of them were described and the chosen metrics' set for safety critical systems was mapped into their internal API. The mapping showed that all three tools are able to measure all of the selected metrics, only the CppDepend does not provide measuring of one of the selected metrics (Response for Class). Also a sample project was analyzed by the tools and the differences in results were discussed. The aim of this part is to identify thresholds of selected metrics and prepare environment for system evaluation.

Although software metrics are well-known and widely used, there is still a discussion about identifying their thresholds and usage. Furthermore, there is a lack of information about software metrics thresholds, particularly about thresholds of safety critical software. This puts evaluators of safety critical systems into a position where they neither have any data for comparison, nor do they know suitable methods for evaluation. This section describes evaluation of software metrics thresholds of one safety critical system and provides information about measured data and data collection and evaluation method. The quality of the system is discussed via determined thresholds in the next section.

4.3.1 Metrics and Thresholds

Using software metrics one can measure the software quality from different points of views, search for potential problems within the software, identify its parts which should be examined or control the whole system development. The broad usage of software metrics is demonstrated by the high number of existing metrics themselves as well as tools used for their measuring [56].

However, the usage of metrics themselves is only part of the software measuring and evaluating process because once we obtain or calculate the values of our specific set of metrics we also need to determine when does the value represent a positive (or at least tolerable) characteristic of the measured software, and when does this value become a sort of warning sign, i.e. an indication of some unsatisfying or unwanted aspect of the software. The boundaries between these two states or sets of values (or any number of shades of grey in between them) are commonly known as thresholds.

Nevertheless, the problem of their actual usage does not lie only in the extensive amount of existing metrics and therefore in selecting the appropriate subset to measure given software. Even if just a limited number of commonly known metrics is used, the result of their application does not have to be unequivocal. Unfortunately, even though there is a considerable effort to make the view on software metrics clear, there still is no widely used and unified view on threshold values of software metrics.

This lack of consistent view is especially obvious in the field of safety-critical software systems. The recently conducted research in this area shows that measuring metrics of different system types brings different threshold values. The results of measuring can be influenced by measurement tool [55], [57], programming language [57], [58], or by the category of the measured software [59].

4.3.2 The Studied System

The system this section concerns itself with is used in railway industry. It is responsible for safe functioning of various devices, in which it serves as a base platform for their own functionality. In a very simplified sense, it can be seen as a hard-real time system, which enables special algorithms to run on their respective devices. The software is written in C++. All software development is being done by experienced team and according to certain coding standards. The whole process of development is being constantly validated and verified. The source code is regularly checked by automatic tools which evaluate compliance with coding standards and compute current values of specified metrics.

4.3.3 The Limitations

According to [58], there are two ways how to determine the software metrics thresholds: by the statistical methods or by the widely accepted threshold values for particular metric. As for our type of analyzed system, these widely accepted threshold values do not exist. Therefore, we find them by using statistical methods.

Statistical methods are based on the principle of derivation of threshold values by collecting data from available systems. Characteristically, we measure various metrics over various systems. Afterwards, we employ statistical methods and derive threshold values. However, this approach has several pitfalls, which have to be considered.

4.3.3.1 General Limitations and Constraints and Their Resolution

Above all, there are differences between evaluated software projects. They can differ in programming language used (Java, C, C++ etc.), in the particular software type (game, office software, operational system etc.), in size (100 LOC – 100 MLOC) etc. Due to this fact, it is necessary to take the programming language, the project type and its size into consideration.

Another significant finding can be seen in [55] where it is shown that different software metrics measuring tools can evaluate the same projects with different threshold values. This is caused particularly by ambiguous definition of software metrics. For example, the computation of the DIT metric (Depth of Inheritance Tree) for Java language can be done with or without including the Object class. As long as different tools are used, the result of the measurement can be different, because each tool can implement different technique of computation.

As shown in [55], the choice of measuring tool can negatively influence the set of classes assigned for inspection. While one tool can identify set A as the set of classes most critical for inspection, another tool can pick set B. In the worst case scenario, the intersection of these sets can be empty. If this happens, the staff responsible for class review can misguidedly define the set of classes for inspection.

Another issue is the large number of existing object-oriented software metrics. It is a common occurrence that for different projects there are often different sets of metrics used for evaluation. We can also see different metrics used for measuring in different studies. This makes it difficult to collect data for analysis and evaluation of metrics thresholds, if we want to compare our measured data with others.

To sum up previous paragraphs, for proper usage of statistical methods for evaluating threshold values of software metrics, we need to have at our disposal only data from similarly typed and sized

software systems coded in the same programming language, measured by the same tool, and with the same set of metrics applied. In case we cannot reach previous requirements, it is probable that the evaluation of software metrics on the grounds of statistical methods is not going to be precise and it could result in misleading conclusions.

4.3.3.2 Specific Limitations of Measuring Safety-Critical Software

As described above, the absence of widely accepted threshold values for the type of systems similar to the one we try to evaluate forces us to use the statistical methods for their determination and therefore we need data from other sufficiently similar systems for analysis and comparison. Unfortunately, in the particular case of safety critical systems we are in an even more difficult situation. There are almost no data published from measuring this kind of software. Safety critical systems are not usually open-source and it is practically impossible to get their source code for measuring. Considering this fact, determination of threshold values of safety critical systems by collecting sufficient amount of data and applying statistical methods is almost impossible.

In this situation, determination of thresholds for safety critical system which is a subject of this study is more than likely to be hard to achieve. It is ill-advised to compare measured values with data from other studies, because there are nearly no results from desirable systems. We also cannot mine data from accessible sites for our own measurement, because there is almost no suitable software for comparison. On the other hand, we still have source code of the analyzed system and tools for measuring. Even if we do not have adequate data for comparison, we can still evaluate our system and determine extremes in measured data. We can apply statistical methods on our data and establish local abnormalities. On the basis of measured data, we can define a set of classes and methods showing unusual measured values which can be afterwards analyzed more rigorously.

4.3.4 Metrics Used

The following metrics were used for threshold selection: CBO, DIT, LCOM, NOC, NOM, RFC, SLOC 1, SLOC 2, and v(g). These chosen metrics correspond to the selected metric's set for the safety critical systems, only the metric CP is missing. The metric CP is missing because it was not included in the quality model of the analyzed system. Simple description of the metrics (retrieved from the help of the tool used for measuring the system) follows, for more detailed information see [60]. The mapping from name into tools API is provided in Table 4.

CBO (Coupling Between Objects)

"The Coupling Between Object Classes (CBO) measure for a class is a count of the number of other classes to which it is coupled. Class A is coupled to class B if class A uses a type, data, or member from class B. This metric is also referred to as Efferent Coupling (Ce). Any number of couplings to a given class counts as 1 towards the metric total"

DIT (Depth of Inheritance Tree)

"The depth of a class within the inheritance hierarchy is the maximum number of nodes from the class node to the root of the inheritance tree. The root node has a DIT of 0. The deeper within the hierarchy, the more methods the class can inherit, increasing its complexity."

LCOM (Lack of Cohesion in Methods)

“100% minus average cohesion for class data members. Calculates what percentage of class methods use a given class instance variable. To calculate, average percentages for all of that class’es instance variables and subtract from 100%. A lower percentage means higher cohesion between class data and methods.”

NOC (Number of Children)

“Number of immediate subclasses. The number of classes one level down the inheritance tree from this class.”

NOM (Number of Methods)

“Number of local (not inherited) methods.”

RFC (Response for Class)

“Number of methods, including inherited ones.”

SLOC1 (Source Lines of Code – Method)

“The number of lines that contain source code.”

SLOC2 (Source Lines of Code - Class)

“The number of lines that contain source code.” For classes this is the sum of the SLOC1 for the member methods of the class.

v(g) (Strict Cyclomatic Complexity)

“The Cyclomatic Complexity with logical conjunction and logical and in conditional expressions also adding 1 to the complexity for each of their occurrences.”

4.3.5 Data Collection and Analysis

The data collection and analysis in here have several phases. The values of the metrics themselves of the studied system need to be collected. After that, some statistics and graphic representation are most likely to be necessary for the data have to be prepared for further analysis. Afterwards, the analysis of the results is performed to categorize the metrics, since further steps of threshold determination may vary in regard to their specific characteristics. Finally, we identify the actual threshold values. The step-by-step description below refers to the way the data from every individual metric is processed in the course of this study.

The metric values measurement through all the classes of the studied system can be easily automated by an appropriate software tool (or tools) and this automation is very desirable as it minimizes the risk of human error. The outcome of this automated tool should include the specific values as well as their graphic representation, most likely histograms, and some descriptive statistics.

Histograms can be used as a visual representation of the data probability distribution (i.e. normal, exponential, etc.) in a form of a graph. In this graph particular for our purposes the x axis values represent possible or observed values of a particular metric for individual classes and the y axis values show the number of classes or methods with the same results.

The usage of descriptive statistics is one of the several possibilities how to evaluate measured data and gain some overall view of the dataset.

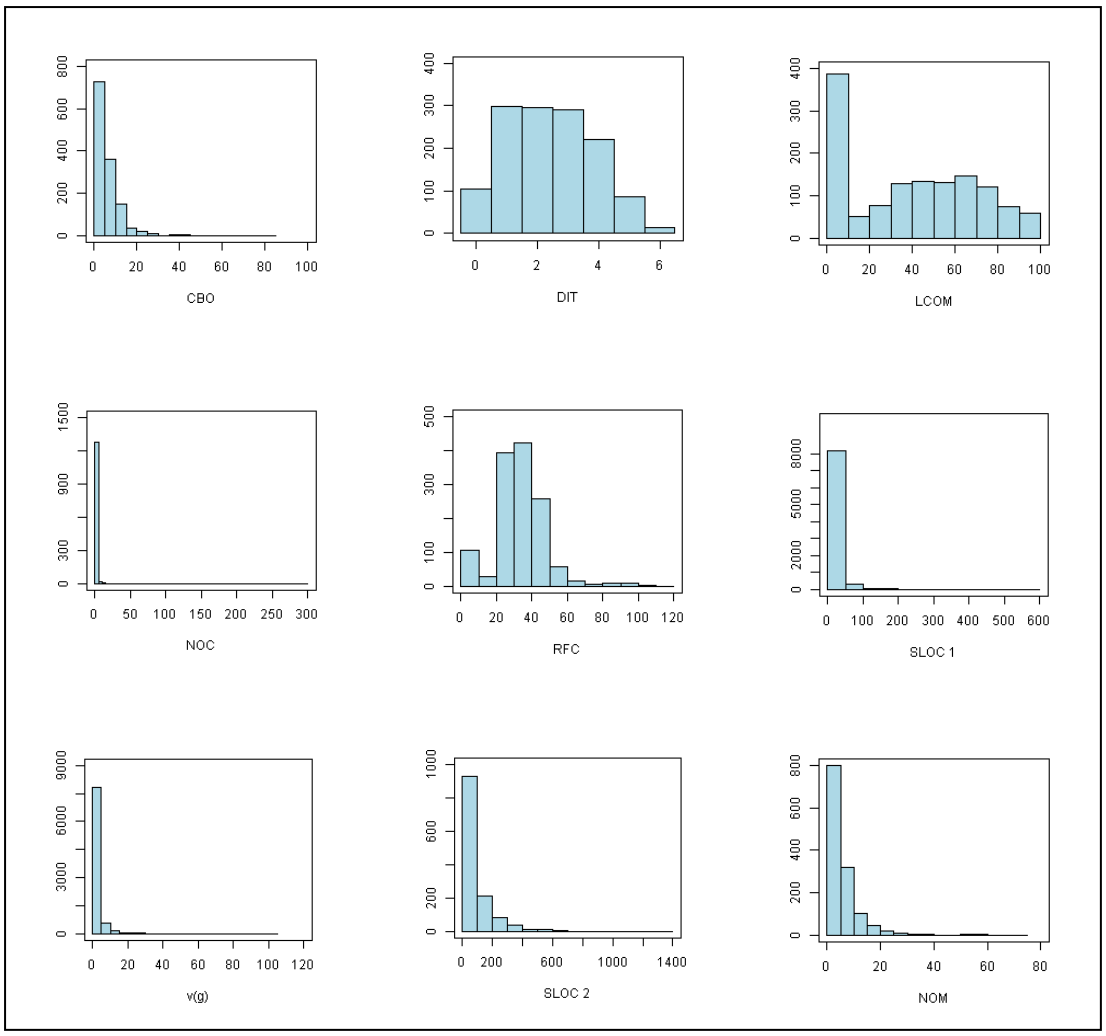

Figure 10. Histograms of measured metrics, the Y axis represents frequency of X values

4.3.6 Measured Descriptive Statistics

The descriptive statistics suitable for the purposes of this study are:

1) Arithmetic mean – represents sum of all the values divided by the number of the values.

2) Standard deviation – shows how significant divergence from the mean exists in the dataset.

3) Modus – the most frequent value in the data set.

4) Median – represents the value located in the middle of the spread of the values.

5) Kurtosis – gives information about whether the data distribution has a peak or not.

6) Skewness – gives information about the symmetry of the data distribution.

7) Minimum and maximum – represent extreme values in the dataset.

The skewness and kurtosis statistics are probably the most difficult to imagine. As an example, normal distribution has the skewness equal zero. Distribution with a positive skewness has longer

right tail and the mean is placed off to the right side of the peak value. On the other hand, distribution with a negative skewness has longer left tail and the mean is placed off to the left side. As we can see in [61], data with skewed distribution can have maximal values considerably different from the mean.

The simplified view on the kurtosis can be “how curvy the final function is”. While the positive kurtosis indicates peaked data distribution, the negative kurtosis can be a sign of a flat data distribution.

4.3.7 Probability Distribution Issue

The so far collected and computed data may not suffice, as there is a different approach to determination of the threshold values with regard to the particular probability distribution that the data from an individual metric show. The analyses of information in studies [27] and [61] show that measured data of some software metrics do not follow normal distribution. Relatively often we can see that measured data approach heavy-tailed distribution and are right-skewed. Normal distribution data should have the skewness and kurtosis around zero [62]. If measured data shows skewness, the distribution does not need to be spread in the close range around the mean value [61]. In this type of distribution, the frequency of high values for random variable is very low whereas frequency of low values is very high [27]. In [61], we can find this information about right-skewed data “This skewness in metrics data affects the interpretation and usage of these metrics in evaluating software quality. Such metrics are not always well characterized by their descriptive statistics such as mean, standard deviation, minimum, maximum, and quartiles“.

As far as we consider this argument, it is necessary to firstly determine the characteristic of measured data and then continue to analyze metric values with regard to the results.

Therefore, before determining the threshold values, we firstly need to categories the metrics according to their histograms and descriptive statistics into two separate groups: the ones showing heavy-tail probability distribution, and the ones with the normal distribution.

With the heavy-tail distributed metrics the approximation and determination of their actual distribution seems appropriate. As described before, the presence of heavy-tail distribution can be observed from the descriptive statistics and histograms but again some automated software tool to help us approximate and determine the actual type of probability distribution should minimize the human error factor, make the whole process more manageable and confirm the observed conclusions.

4.3.8 Data Categorization

Finally, after the categorization of metrics through their respective probability distribution, we should have all the input data necessary for the actual threshold value determination.

In [58], a range of typical values is used for metrics evaluation. This range represents typical values of particular metrics for which its lower and upper boundaries and extreme values are defined. For computation of typical values and upper and lower boundaries the arithmetic mean and deviation are used. If a measured value of a metric is 50 % higher than the highest value of the interval between upper and lower boundary then the value is evaluated as an extreme value. This approach is suitable in the case where the data have normal distribution. As data with normal distribution are centralized around their mean values, we can look at their values as determinative.

In [27], the technique of separating values into three categories was used for assessing the thresholds values of heavy-tailed data. The categories are assigned the names of “good”, “regular”, and “bad” and each category represents particular range of values.

The “good” category represents range which contains the most frequent values. The values with low frequency of occurrence but which are still considered as not being rare belongs to the “regular” category. The “bad” category contains values, the occurrences of which are rare. This technique is appropriate for our heavy-tailed data as well.

Applying the above mentioned process on safety critical system will result in getting the set of metric values of each class and method. By selecting the classes or methods, values of which cross threshold values, we get set of classes or methods fit for review.

4.3.9 Tools Used

For the analysis, three tools were used. First, the tool Understand C++ 3.1 [52] was used for measuring all the metrics values of the studied system. We chose this tool because it was requirement from the company developing the system.

Second, for simplification and it´s widely usage, the Microsoft (MS) Excel 2003 was chosen for computation of the descriptive statistics and selection of classes for inspection.

Ultimately, the EasyFit tool (www.mathwave.com) was used for the analysis to fit the measured data to various probability distributions. This tool was chosen because of its simple usage and huge amount of probability distributions it can evaluate.

4.3.10 Execution

Here we sum up the actual process of data collection and analysis in the course of the study execution.

The data from source code were gathered by Understand C++. For any measured metric its own MS Excel file with results was created. After all metrics were measured, The MS Excel functions for computing of all the descriptive statistics were used. For gaining better insight on data representation, we created a histogram for each measured metric. If descriptive statistics and histograms evinced heavy-tail data distribution, the EasyFit tool was used for data distribution fitting. In the end, MS Excel was used for selecting classes for review.

4.3.11 Results Presentation and Further Analysis

As we can see in Table 8 and Table 9, all metrics except RFC and LCOM have mean >= median >= mode. This can be seen as a sign of the right skewness in the data [61].

The peaked characteristic of data shows the positive kurtosis in all the measured metrics. Metrics CBO, NOC, NOM, SLOC1, SLOC2, and v(g) have mean values notably smaller than maximal value, which also shows the skewness of data distribution. The kurtosis values for DIT and LCOM are negative, so we can consider DIT and LCOM as having a flat distribution.

	CBO	DIT	LCOM	NOC	NOM
Arithmetic mean	6,28	2,41	39,64	0,92	6,42
Median	5	2	41	0	4
Modus	2	1	0	0	3
Standard deviation	6,16	1,43	31,57	8,22	6,8
Kurtosis	22,62	-0,71	-1,26	932,91	23,99
Skewness	3,14	0,21	0,08	28,62	3,96
Minimum	0	0	0	0	0
Maximum	82	6	100	274	75

Table 8. Measured descriptive statistics of CBO, DIT, LCOM, NOC and NOM

	RFC	SLOC 1	SLOC 2	v(g)
Arithmetic mean	33,85	14,92	94,74	2,9
Median	33	7	55	1
Modus	39	5	19	1
Standard deviation	14,91	26,67	120,8	5,47
Kurtosis	3,19	116,13	20,37	94,53
Skewness	0,66	8,56	3,62	8,01
Minimum	0	1	3	1
Maximum	106	582	1361	103

Table 9. Measured descriptive statistics of RFC, SLOC1, SLOC2, v(g)

In the given histograms the right skew of the values of CBO, NOC, NOM, SLOC1, SLOC2, and v(g) is noticeable as we can see high concentration of values in the left side of the histograms. The visual observation confirms the results of descriptive statistics, whereas for the given set of metrics mean >= median >= mode and also relatively high values of the skewness and the kurtosis exist.

Different results can be found in the data from DIT, LCOM and RFC. The data from DIT do not show any concentration of values on the right side of the histogram. The shape of their histograms and also the results from the descriptive analysis evince the similarity with normal distribution (negative skewness and low values of kurtosis).

The LCOM data have a peak at zero; the rest of the histogram corresponds with the normal data distribution. The descriptive statistics suggest the LCOM data to be flat and non-skewed (the mean >= median >= mode does not hold here).

We can see high concentration of values around the mean in the RFC histogram. The results from the descriptive statistics show that the RFC data have relatively low values of the skewness and kurtosis. In the RFC data, the relation mean >= median >= mode is not valid.

For these reasons, we will not analyze the data from these three metrics (DIT, LCOM and RFC) with regard to the existence of heavy-tail. The threshold values for these three metrics will be established from the mean and the standard deviation will be established according to the method used in [58].

4.3.11.1 Heavy-Tail Analysis

The existence of heavy-tail was analyzed for the data from metrics CBO, NOC, NOM, SLOC1, SLOC2, and v(g). To determine the best probability distribution, we considered results from the EasyFit tool and the visual representation of the data probability functions. In Figures 11, 12 and 13 we can see the results of fitting function.

The data from the CBO metric can be characterized by generalized Pareto distribution. This data distribution belongs to heavy-tail probability distributions [63].

This distribution also fitted data from NOM and SLOC2 metrics. The Weibull distribution characterized very well the data of the NOC metric. The Weibull distribution also belongs to heavy-tail probability distributions [27]. The data from the SLOC1 metrics were well fitted by the Lognormal distribution, while the data from the v(g) metrics fitted the Exponential distribution. Both of these distributions are considered to be heavy-tailed [64].

From these results, we can see that all the data from the metrics CBO, NOC, NOM, SLOC1, SLOC2, and v(g) can be approximated by heavy-tailed distributions. According to [65], if data are heavy-tailed, their mean is not representative. Because of this, it is not convenient to use mean for establishing the threshold values. As a result, the thresholds values for these metrics were assessed according to the method used in [27] by separating values to three categories.

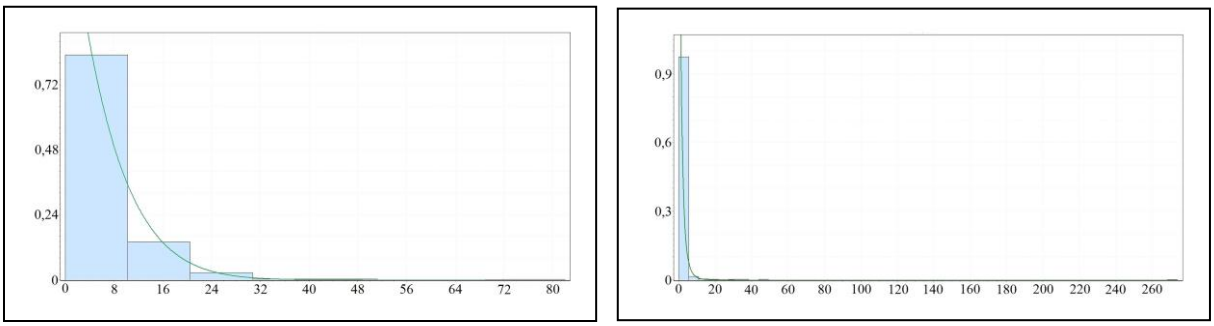

Figure 11. CBO with generalized Pareto distribution and NOC with Weibull distribution

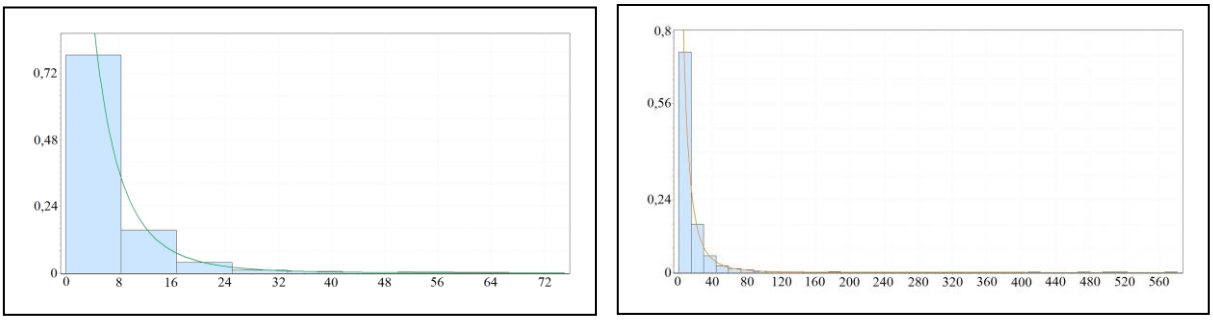

Figure 12. NOM with generalized Pareto distribution SLOC1 with lognormal distribution

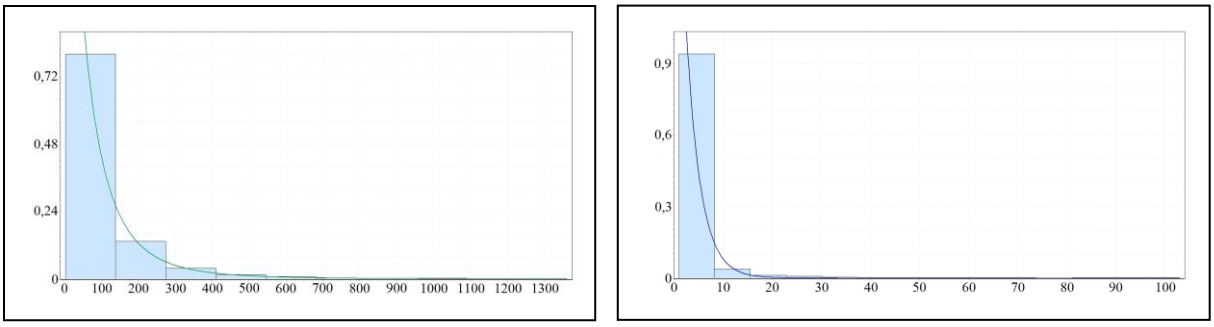

Figure 13. SLOC2 with generalized Pareto distribution and v(g) with exponential distribution

4.3.11.2 Identifying Thresholds Values for Metrics with Heavy-Tail Data Distribution

We used the method form [27] for identifying thresholds values from our data. We established three categories for the metrics CBO, NOC, NOM, SLOC1, SLOC2, and v(g) with the names "good", "regular" and "bad". The values for these categories were drawn from data and histogram analysis.

The values of the CBO metric are most frequently spread in the range 0 – 15. Almost 90% of all values were lower than 10. It is clear to see from the plots that values between 16 and 25 were presented much less frequently, but still with better probability then values higher than 25. For the CBO metric the value ranges for the categories were designated as: 0 – 15 good, 16 – 25 regular, 25+ bad.

The most frequent value in the NOC metric is 0. From the definition of the NOC metric, this value represents classes with no children. There are almost 95% of all classes without children. Classes with 1 to 5 children occur much less frequently, but we can still see their presence in the plots. The probability of the occurrence of the class with more than 5 children is very low. For the NOC metric we assigned the three categories as: 0 good, 1 – 5 regular, 6+ bad.

The NOM metrics has the most values in the range 0 – 10. Almost 90% of all the values are positioned in this range. The values in the interval 10 – 25 are presented with lower frequency, but are still apparent in the plots. The values higher than 25 can be considered as rare. We designed the three categories accordingly: 0 – 10 good, 11 – 25 regular, 26+ bad.

The values of the SLOC1 metric are most frequent in the range 0 – 10. We can find approximately 90 % of all the values in this interval. To the next category, we can put the values in the range 11 – 69. The values higher than 69 can be find very seldom. We established the three categories as 0 – 10 good, 11-69 regular, 70+ bad.

The SLOC2 values are most frequently spread in the interval 1 – 200. The values can be found in this range with almost 95% probability. The values 201 – 400 are still evident in the data, but not so often as data from the previous category. The values over 400 are rare. The three categories are therefore: 1 – 200 good, 201 – 400 regular, 401+ bad.

The most frequent value in the data from the v(g) metric is 1 and probability of this value is almost 70 %. Almost 25 % of all the values from this metrics are in the range 2 – 15. The values over 15 occur very seldom. For the v(g) metric we established the three categories as: 1 good, 2 – 15 regular, 16+ bad.

4.3.11.3 Identifying Thresholds Values for Metrics without Heavy-Tail Data Distribution

The data from the metrics DIT, LCOM, and RFC do not evince heavy-tail distribution. To identify thresholds values for these metric, we used method showed in [58]. For each of these metrics we used results from the descriptive statistics – the mean and the standard deviation – for calculating the threshold values. The first threshold value corresponds with the mean and represents the most typical value. The second threshold value is calculated as a sum of the mean and the standard deviation. This second value represents high, but still acceptable set of values. The third threshold value is calculated as a multiplication of the second threshold value by the coefficient 1.5 [58]. The third threshold value is considered as an extreme and should not be present in the dataset.

The typical value (mean) of the DIT metrics is 2, the second threshold value corresponds to 4 and extreme value is 6. The LCOM metric has the typical value equal to 40, the high – but still not extreme – value is determined as 72. We calculated the extreme value of LCOM as 108. However, as we can see in the histogram of LCOM measured values, there is no class with the LCOM metric higher than 100. This result is influenced by the special shape of the data from LCOM metric. As can be seen in the histogram of LCOM, the data are concentrated in the low values, but with higher values, they start to have characteristic of normal distributed data. Because of this behavior, the extreme values were identified manually as for the heavy-tailed metrics data and extreme for LCOM metrics was set as 100. For the RFC the typical value is 34, the high value is 49 and the extreme value is considered to be 74.

4.3.12 Identified Thresholds

Data for CBO, DIT, LCOM, NOC, NOM, RFC, SLOC1, SLOC2, and v(g) metrics were collected with the use of the Understand C++ tool. The system analyzed does not correspond by its function and robustness to free and accessible application. Because of this, we could not compare the measured data with the data from other published studies nor with the data measured from any free-to-download software from the internet. In addition, the data from similar software are almost impossible to gather. For all these reasons, we did not use statistical data from any other previous studies.

We analyzed the measured data for the existence of heavy-tail and according to results, we categorized our data into two groups. The first group contains metrics with potential heavy-tail data distribution, the second without it. With regard to the results from the descriptive statistics and histograms, we assigned the CBO, NOC, NOM, SLOC1, SLOC2, and v(g) metrics to the first group, leaving the DIT, LCOM, and RFC metrics to be classified into the second group. For both groups the thresholds values were established. For the first group, we identified the threshold values from the data and the plot characteristics by using method from [27]. The threshold values for the second group were determined from the descriptive statistics with the method used in [58]. The results are shown in the Table 10.

	Heavy tail	Good / Typical value	Regular / High value	Bad / Extreme value
CBO	yes	0 - 15	16 - 25	25+
DIT	no	2	4	6
LCOM	no	40	72	100+
NOC	yes	0	1 - 5	6+
NOM	yes	0 - 10	11 - 25	26+
RFC	no	34	49	74
SLOC 1	yes	0 - 10	11 - 69	70+
SLOC 2	yes	1 - 200	201 - 400	401+
v(g)	yes	1	2 - 15	16+

Table 10. Identified thresholds of measured metrics. The good, regular and bad are valid for heavy-tailed data

4.4 Evaluation of System Quality

In the previous part, the metrics' thresholds of the safety critical system were identified. In this part, the aim is to evaluate the software system in "the light" of determined thresholds values. We used MS Excel files with measured values and MS Excel functions for identifying classes and methods, which should be reviewed, because they evince abnormal values.

We aimed only at the category "bad", because classes or methods with the worst evaluation could be the most harmful for the system. For the CBO metric, we identified 24 classes, which were in the category over 25. The NOC metric showed that 27 classes should be reviewed. Similarly, the NOM metrics detected 25 classes for review. Most classes for review showed the SLOC2 metrics – 35. From the non-heavy tailed metrics, the DIT identified 15 classes for review, while RFC identified 24. The metrics specialized in methods also detected possible problems – the v(g) metric detected 187 methods while the SLOC1 metric showed 212 methods.

The classes and methods with measured metrics values over the specified thresholds identify the possible quality weaknesses in the measured system. All these classes and methods were reported as possible threats and passed to the persons responsible for system quality assurance. On the other hand, classes and methods values of which do not cross the threshold value can be seen as non-urgent candidates for inspection, especially the classes / methods which results fall into the good / typical values. Together, these two sets of classes and methods give information about the measured system quality.

Metric	Classes / methods identified for inspection
CBO	24 classes
DIT	15 classes
LCOM	10 classes
NOC	27 classes
NOM	25 classes
RFC	24 classes
SLOC 1	212 methods
SLOC 2	35 classes
v(g)	187 methods

Table 11. Numbers of classes and methods for inspection after performing evaluation of system quality

5 Improving Quality of Safety Critical Systems

In the previous section, software metrics relevant for safety critical software were selected. Software metrics' thresholds for one particular safety critical system were defined and based on these thresholds, the system quality was evaluated. In this section, we will focus on a particular method which aims at improving quality of safety critical systems. The idea behind this is that improving system according to important metric brings more benefits then improving system according to not so important metric.

5.1 Improving Software Quality According to Software Metrics

Some metrics can be more significant in measuring software quality sub-characteristics then the others. The ISO 9126 defines its quality model and proposes quite wide set of software metrics which can be used for measuring sub-characteristics. The set is not exhaustive and user can arbitrarily enhance the metrics by his or her own metrics and widen the whole metrics suite. Important question in here is the significance of particular metrics used in the quality model. The ISO 9126 was criticized by several authors [33], [66] for the lack of metrics weight, clarity, etc.

Improving system according to important metric can influence the overall quality of the software system more than improving the system according to a metric which does not have that much importance. User can get results from measurement of the system and some metrics can have their values over the thresholds. In this situation, with limited budget, time and experience, it can be problematic to select which part of the system should be improved first. Especially, if more metrics evince values over or close to specified thresholds, user has to select which metric should be improved. If user selects a metric with biggest importance and improves the system according to the measure of this metric, the quality of the system should improve more rapidly in comparison with improving system according to less important metric.

5.1.1 Important Metrics for Quality Improvement

The quality of software system can be improved more quickly if the system is improved with focus on more important software metrics. At the moment, there is no clear answer to the question how to define importance for particular metric, although several authors touched this topic in their work [14], [32], [33]. The importance in the cited works is often specified based on gathered experience of the author or his colleagues.

Testing of safety critical software is very important, because as it was discussed above, failure of this kind of system jeopardize human life and / or can produce huge financial loss. Testing is one of the means to prevent system from the problems before its deployment into operation. For example, the importance of testing of safety critical system is pointed out by the norm ISO 50128 [36] for railway industry. This norm defines strict criteria for testing which need to be fulfilled to make the system approved for the usage in the railway industry.

Because the term testing is wide and defines several activities, there exists quite a wide set of metrics, which can be measured. For example, in recently published reports [67] and [68] we can find over 20 metrics to be measured in the area of testing. One of common metrics to measure in testing is the test coverage metric. In the area of safety critical systems, the test coverage metric is very important – for example, NASA in [69] says about safety critical systems: "These systems may require a formal, rigorous program of quality and safety assurance to ensure complete coverage and analysis

of all requirements, design, code, and tests." Another example from the safety critical area is the norm DO-178 Level B used in airborne industry [70], which defines strict demands about verification of the software code, specifying for each level of safety particular degree of test coverage.

As can be seen, the test coverage is important metric and is widely popular not just in safety critical software area. The importance of the metric is highlighted in several safety-critical norms and we can consider the metric as widely known in common. Because of its importance, we can say that by improving test coverage we can improve the values gathered by measuring the test coverage metric. By improving values of the metrics we will improve the whole quality of the system.

Important question at this point is which kinds of means do exist to improve the test coverage. As one of the possible answers to the question, the method of test inputs generation via symbolic execution can be considered. This method enables huge improvement of results of the test coverage metric by generating tests which tends to cover most of the program paths. Because this method can be very useful for improving the quality of the software system, it will be described in the following subsection.

5.2 Method of Test Inputs Generation via Symbolic Execution

Usage of the test input generation via symbolic execution method can produce wide set of tests. Due to character of the method, applying these tests to the system can reduce the possible amount of errors in the system and improve the results of the test coverage metric. The popularity of this method is confirmed by considerable amount of articles, developed tools and patents. Although the method of test input generation via symbolic execution can be used regardless of programming language, we will focus on the C++ language and apply the method to the code of one particular safety critical software.

To get proper understanding of the method and how it can help to improve the software quality, firstly, the area of software testing will be discussed. Several test inputs generation methods will be assessed later. Afterwards, the general method of symbolic execution will be described and then the focus will be targeted on the symbolic execution of C++ code. As there is no direct tool enabling symbolic execution of C++, we introduce comparison of three tools, which could be used for it with some limitations.

5.2.1 Software Testing

People are making errors every day – error can be wrongly adjusted time on an alarm clock, badly fasten seat belt, wrong punctuation in a letter etc. People usually do not want to make these errors, but they are part of our lives. In software, we are facing errors on daily basis. Programmers usually write bugs unintentionally just as ordinal people make grammar errors in the letters. However, some errors in the software can be critical. Execution of a program code of which code contains errors can lead to the failure of the whole application. This failure can be seen as an unexpected behavior of the application. The more the software can influence human life, the more vigorously we should try to prevent its code from the errors. To be able to prevent the software from the failures, we need to check it for the unexpected behavior. The software testing is a process of determining whether the tested program behaves according to expected behavior [71].

Software errors cost U.S. economy upwards to 60$ billion every year [72]. This amount of money is spent on fixing software although it could be used in other more useful ways. The importance of

testing could be seen in recent accident with HeartBleed bug in OpenSSL [73]. The thorough negative testing was pointed as a possible instrument to avoid this kind of error long before its discovering in April 2014 [74]. The software testing constitutes inseparable part of the software engineering and all prudent companies are investing into the software testers, because this investment can spare a lot of money in the future. Nevertheless, even if the software testing is perceived as the number one technique in the war with software errors, many companies still lack the quality software tests and testers. The reason is simple, the software testing is time and resource consuming task. According to [75], the software testing requires up to 50 % of the total development cost and the price can be even higher in the area of the safety critical systems. In addition to the high cost, there are usually no tangible artifacts for the management, because testing does not give any visible attribute into the software. When negotiating with blindfolded investor, it is simpler to justify the investment in the new software feature, than in the new integration tests. Therefore, one possibility to broaden the usage of software testing is lowering its cost.

To lower the cost of software testing, the automation techniques are being developed. The automated test tools can facilitate the work of software engineers, quality assurance staffs, project managers etc. The area of software testing is complex and involves many procedures [76], so there are different kinds of tools for test alleviation. One of the testing procedures, where the testing tools are aiming, is the automatic generation of the test cases. Writing test cases manually is often demanding and long-term job. To achieve the quality test coverage requires deep knowledge of the tested system and hard-labor work of the testers. For a skilled tester, even a small web application with a simple input form [77] can cost several days of writing tests for at least majority of all possible user inputs. In case of the more robust and complex system, for example the safety critical system, writing comprehensive tests manually is almost impossible. To make the process finding the right test input data for the test cases simpler, the automatic test input generation techniques can be helpful.

5.2.1.1 Testing Safety Critical Software

One area, where safety critical systems are essential, is the railway industry. Because of its huge responsibility, the software in the safety critical systems in the railway industry must comply with demanding criteria before its deployment into the real operation. The requirements for these systems are written down in norms. One of them is the norm EN 50128 from CENELEC (European Committee for Electrotechnical Standardization), which defines "process and technical requirements for the development of software for programmable electronic systems for use in railway control and protection applications" [36]. This norm, beside other things, touches the process of testing and quality assurance. According to this norm, the railway safety critical system which is SIL4 (Safety Integrity Level) [44] should make relatively thorough testing to prevent it from errors. The requirements for testing are demanding and can considerably enlarge the company's overall budget. Therefore, any tool that makes the process of testing simpler is warmly welcomed in the software departments of railway industry.

5.2.1.2 The Analyzed Real-Time Safety Critical System

The Department of Computer Science and Engineering of the University of West Bohemia is collaborating with the major Czech company involved in railway industry. One part of collaboration is the process of rigorous testing of the railway-safety critical software. The software is generic real-time safety critical system, which is responsible for functioning of various types of railways devices.

Its main mission is to provide a base platform, where the software of these devices can run. In a simplified view, the system can be seen as a specialized operational system, which enables running of algorithms of real devices. The system is implemented mostly in C++ under MISRA C++ 2008 rules [78]. The system contains approximately 1000+ classes and approximately 190 KLOC.

The system has been tested mostly from the functional point of view. There are many tests, where the right functioning of the whole system is being examined. During design and implementation of the system, there was strong emphasis on modular architecture. Consequently, there are many tests examining the modular architecture and inner modules communication. As the system forms a base for other devices, there are also many tests examining the integration properties. Also a set of unit tests was implemented before but on the account of complexity and size of the system, these tests mainly aimed at the critical parts of the system. The broadening of unit tests could enlarge the test coverage of the code, improve the overall quality of the system and improve the compliance with the norm.

5.2.2 Test Inputs Generation Techniques

The manual creation of test inputs is an intensive and hard work. The problem of test inputs generation exists for a long time, so there are many research techniques focused on its improvement. We will describe four of them – three here and the fourth technique – symbolic execution – in the next section. Although there are other techniques (mutation testing, search based testing etc.) we will not cover them, because it goes beyond the scope of this work.

5.2.2.1 Random Testing

The idea of testing software randomly lies in generating random inputs, execution of the software with them and comparing, if the program behaves correctly (for example, methods contracts are met or program does not crash). This approach of the testing is simply in concept, widely used and usually easy to implement [79]. This technique is often used as a supplement for more systematic techniques.

The main drawback of the random testing lies in generation of huge number of very similar test inputs which do not achieve high code coverage. There are debates about the efficiency of random techniques. Several empirical studies showed random testing generation technique achieves less code coverage then systematic techniques [80], [81], [82]. Although the simple random testing probably cannot compete with systematic techniques in code coverage, there are many alternation techniques, which can make it quite useful. Five of them are described in [83]. These adaptive random testing techniques can enhance the input generation and improve the code coverage. As an example, the feedback-directed random test generation [84] achieved comparable results with the systematic approaches [85].

5.2.2.2 Combinatorial Testing

This testing technique has two main domains of usage – the configuration testing and the test inputs generation. In both domains, the technique trims the possible input space to necessary minimum and use this reduced space for the test generation. It can reduce the number of tests. The principle lies in the premise that the most bugs can be triggered by one or two parameters combination. Bugs requiring three or more parameters interaction are progressively less common [86]. According to the studies described in [87], the amount of bugs triggered up to two combinations of the inputs is between 70% and 95% dependable on the type of the project.

The combinatorial technique can theoretically generate high quality tests at lower costs. The only problematic point is its usage for the generation of software methods inputs, where the parameters can be integer or float types, which can have infinite number of combinations. There are also tools [88], [89] etc. that facilitate the combinatorial testing. An understandable example of how the combinatorial testing works can be seen in [90].

5.2.2.3 Model-based Testing

Model-based testing is based on the concept of deriving the test cases from the model of the system under test (SUT). Because the tests are derived from the model, this technique can be seen as a black-box testing. In the view of the model-based testing, the SUT is a black-box system, which accepts inputs and produces outputs. Its inner state is changing due to the input/output processing [91]. The model of the SUT contains information about the input and the output sequences on specified level of SUT abstraction. The test suites are derived from the model by selecting a finite subset of the sequences and they check the conformance of the model with the program.

The main characteristic of the model-base testing is the existence of the model of the system. To use the model-based testing properly, one need to have the model of the system or to create it from the scratch. Because the creation of a model in the middle of development can be problematic and tedious, the model-based testing techniques suit better the development processes which naturally contain the model.

5.2.3 Symbolic Execution

The idea of symbolic execution lies in making the program inputs symbolic and in executing the program with these symbolic values instead of the concrete ones. Unlike in the concrete execution, in the symbolic execution the program can take theoretically any program path. During the execution with the symbolic values, the path constraints (PCs) are collected over the symbolic values. The PCs represent conditions under which the program flow can reach a concrete program path. After the symbolic execution finishes, by using the PCs, one can compute the concrete values. Solution of these PC can be used as test inputs that make the program use the correspondent path. The technique can be used for generating tests with extensive test coverage, because by it's application one can derive high percentage of the possible paths through the program.

PC is a Boolean formula, accumulating all current constraints over the symbolic input for one path through the program. Every branch condition makes the PC updated and creates another PC. Basically, for the "then" branch another PC (PC´) is created, while the "else" branch updates the original PC. After the branch condition is evaluated, both PCs will be inspected for satisfiability, which is done by the constrain solver. If any PC becomes unsatisfiable, the symbolic execution will not continue along the corresponding path and the path is marked as unfeasible. Otherwise, the symbolic execution will continue. The symbolic execution ends when reaches an exit statement or triggers an error (assertion violation etc.). The symbolic execution engine also needs to handle the program state, which maps variables to symbolic expressions and the program counter, which points to the next executed statement.

Simple example of a code examined by symbolic execution can be found in Figure 14. The code executed by the symbolic execution is represented by the symbolic execution tree. Nodes of the tree represent program states, while the edges represent states transitions. First line of each node describes the mapping of variables into symbolic values. The second line contains the PCs. In the

beginning, the two variables x and y are made symbolic by assigning them into symbolic values. After the line 2, the program splits into two branches, each branch has its own PC. The "else" branch directly ends, while the "then" branch continues by updating the x. The next split is after the line 4 is executed and another two branches are created. As the symbolic value of x has been updated before, the PCs of both branches contain this updated value. In the end of the symbolic execution, three paths through the program code were revealed. Each path is described by its own PC. As the next step, the PCs from all the paths are used to generate the test data. In our example, the 2,3,4,5,8 paths can be tested by the input -100 and -50. The paths 2,3,4,8 can be tested by the input -10 and 1. The test input 100 and 50 will cover the remaining paths 2,8.

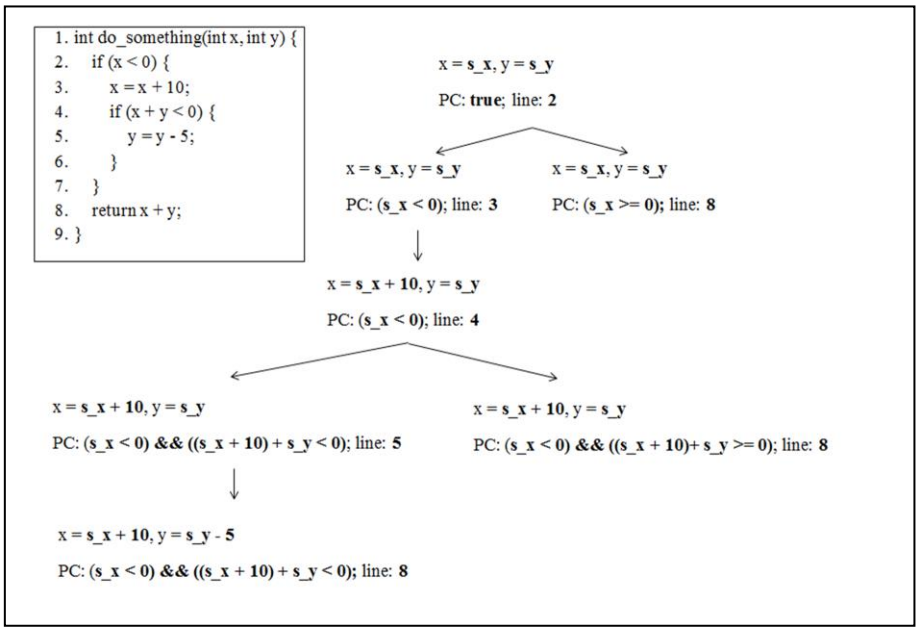

Figure 14. Symbolic execution of a simple program

The symbolic execution is known for almost forty years. First publications are dated from the mid-1970s [92], [93], [94] and [95] (most cited paper). Although the technique is relatively old, it registered larger expansion in the research area just recently. There are at least two known reasons for this. Firstly, the symbolic execution strongly depends on the constraint solvers. The more efficient solver the symbolic executor has, the more complex constraints can be evaluated. There are nowadays many constraint solvers [96], [97], [98] which can be used for the purpose of the symbolic execution. The second reason for the current progress is the huge computing power developed in last decades. For example, in comparison with mid-1980s, the Apple iPad 2 is as fast as the Cray-2 [99]. The symbolic execution can be extremely expensive if it traces the paths through the program code, so better power is appreciated.

5.2.3.1 Challenges of the Symbolic Execution

Although the symbolic execution is popular research area of computer scientist, there are still many challenges, where further research is needed. There are many tasks where some extra exploration is required but because of the limited scope of this work, just a few of the most important will be shortly mentioned in the next paragraphs. It is needed to point out that the problems of symbolic execution are frequently highly tied together. For example, improvement of a constraint solver by

widening its functionality for new operations can deteriorate the whole executor power, because it will need to explore more paths.

1) Extern Calls

We define extern calls as a program calls to functions whose results cannot be retrieved while the program is being symbolically executed. While the program is symbolically executed, we cannot know the result of calling these functions unless we also symbolically execute them. Majority of present programs contains many calls of external libraries, external functions, system calls etc. Therefore, there is a need to solve this problem.

At the moment, several solutions have been presented and verified. One of them is to replace the unknown function call by random value, where the type of the value is determined according to the returning function type. Another solution is gathering the results of external functions call from concrete execution of the program. These approaches are used in so-called concolic (concrete + symbolic) execution [100], also known as white-box fuzzing [101] or dynamic symbolic execution [102]. Concolic testing can handle most of the extern calls but we pay for the concretization by possible false-positive reports.

2) Path Explosion

Even a small program with a few lines can contain huge amount of possible paths. With the increase of the length of an execution path, the number of the feasible paths increases exponentially [103]. To reach all possible paths can be, even with current computer power, hard to realize and with modern complex software, it becomes almost impossible. The path explosion problem is seen as the main challenge in the symbolic execution [104].

Although the simple and robust solution is still nonexistent, many approaches alleviating this problem were discovered. Several of the techniques are based on intelligent search strategies during the path analysis – for example, the generational search [105] or the combination of coverage-optimized search and the random path selection strategy [106]. Another approach for facing this problem can be improving the loop navigation [107] or taking the random paths during the execution process [108].

3) Complex Constraints

Solving constraints is a NP-completed problem [106] and it makes the symbolic execution engines spend most of its time in solving complex constraints. Recent constraint solvers are able to handle a large amount of constraints, but still have limitation in non-linear arithmetic operations such as the multiplication, division, mathematical functions, pointer operations, floating number arithmetic etc.

To facilitate the work of constraint solvers, several techniques were presented in last years. For example, the caching technique works on principle of saving constraints and their results and afterwards comparing the current processed constraint with the cached ones. If we hit the cache, we save time of solving constraint [109]. Other techniques use irrelevant constraint elimination [99] or concretization [106].

5.2.3.2 Symbolic Execution of C++ code

The C++ language has been developed a lot in last years and today with its new standards and features represents modern and heavily used programming language for the software developers. Because of richness of its mechanisms (constructors, destructors, operator overloading, multiple inheritance, templates, exceptions, lambdas, functors etc.) it is a hard tasks to parse it. The C++ grammar is ambiguous and while parsing the code, the meaning often depends on its context [110]. This demanding character of the C++ mostly caused the lack of tools for symbolic execution of this language. On the other hand, distinct part of industry software is written right in C++ which makes it interesting for researches from the industry area.

The C++ language is based on the C language. Many articles were written about symbolic execution of the C code and there exists significant number of tools allowing symbolic execution of C, while there are just a few articles about this technique with C++ and no public available tool for the C++. We need to note that the tools for the symbolic execution of the C code often propose also symbolic execution of the C++, but the reliability is usually not sufficient for real usage.

Although currently, there is no public available tool, there are several articles describing successful implementation of the symbolic execution of the C++ [111], [112]. The tool KLOVER [112] implemented by Guodong et al adapted the tool KLEE [106] to work with the C++ code. KLEE is a symbolic executor which uses the LLVM compiler infrastructure [113] to alleviate the program analysis and generate test inputs for the program. To achieve sufficient functionality when handling with C++, they had to make KLEE work with new instructions. As an example, they implemented the llvm.eh instructions to handle exceptions. Another adaption was done on the uClibc++ library [114], which was used as a substitution for the standard C++ library. This adjustment brought considerable speeding up of symbolic execution and also strongly improved the code coverage. The tests on public available C++ programs showed mostly better code coverage with much less tests required in comparison with manual testing. The same was achieved in real-world industry application, where the test driver extent was smaller than the one in manual testing [112].

The tool [111] presented by Garg et al uses a principle of combination of concolic testing with alleviated random testing. Their approach is named as “Feedback-directed random class unit test generation using symbolic execution” and is patented [115]. For the processing of the C++, their own intermediate language (IL) is used. To compile origin source code into the form of IL, they used their own tool called CILpp, which is based on the CiL tool [116]. To generate tests, they firstly try to reach all possible paths by alleviated random inputs generation adopted from the tool Randoop [85]. When the code coverage stops improving, the concolic execution is utilized to enhance the test generation. The executed tests on publicly available software proved overall quality of this concept by increasing the code coverage in a compartment with separately usage of the techniques. Unfortunately, neither the CILpp is public available, nor is the whole tool.

Although there are no available symbolic executors for the C++ code recently, the future seems to be optimistic. There are a few promising projects, which could move the current state ahead. One of them could be the “Native Symbolic Execution” [117] which uses “lightweight KLEE style symbolic execution technique”. Another project, which aim is to develop comprehensive toolbox for program analysis, is the STANCE project [118]. Albeit the aim of the project is not directly the symbolic

execution due to its wide extent for developing the toolbox, we can assume it will also have positive effects on the symbolic execution of the C++.

As the area of the symbolic execution is wide and the tools are still developed and new articles are constantly accumulated, there are some web pages, aimed on the news tracking. On one of them [119], is listing many testing tools, and some of them are determined for the symbolic execution. Another page [120] is dedicated to track only the "code based test generation tools". Further sources of information can be obtained by the comparison of articles [121] and [122].

5.2.4 The Tools Comparison

As we mentioned in the beginning, we will compare three available tools for the symbolic execution of the C++ code. We picked up these tools, because all of them are open-source and they can with limitation carry out the symbolic execution of the C++. We also need to note that all the three tools are connected with the LLVM compiler infrastructure [113].

5.2.4.1 Clang Static Analyzer

Clang [123] is a modern, open-source and popular compiler for the C/C++ code as for the Objective-C and Objective-C++. Clang uses LLVM as its backend, so it enables to compile into the LLVM intermediate language. Its popularity is growing as the companies like Apple, Google, ARM etc. are involved in its development. According to the tests from the middle of 2014 [124], the Clang beats the GCC in the performance in more than 1/3 of benchmarks.

The Clang Static Analyzer (CSA) [125] is a part of the Clang infrastructure and is used not only by the Apple developers to check the code for bugs (for example, in this article the CSA is used to find the HeartBleed bug [126]). As the Clang, the CSA is also open-source and can be used by anyone, who has installed the Clang. The CSA performs path-sensitive exploration of the program and passes over the processed information about the code to so-called "checkers". Every checker is usually responsible for handling one type of errors, for example, one checker takes care of division by zero, another one of null pointer dereference etc.

One of the key benefits of the CSA is the possibility to write own checkers. For example, the project [127] uses the CSA for writing the MISRA C++ compliance checkers. The openness and good documentation makes relatively easy to write own checkers. To implement a checker, one has to extend the template class Checker and overwrite some of its functions. One of these functions is "checkPreCall" – every time the CSA realizes the code continues with a method, it calls all the checkers which have overwritten this method with the information about the current state and the method details. Every checker can then react as it thinks fit. An example can be seen in the Figure 15. In the example, the checker will print the text "Hello main" every time before the function with the name "main" will be called by the analyzer core.

```
1. class CheckerExample : public Checker<check::PreCall> {
2. public:
3. void checkPreCall(const CallEvent &Call, CheckerContext &C) const {
4.       if (C.getCalleeName(CE) == "main") { llvm::errs() << "Hello main."}
5. }};
```

Figure 15. Example of a simple checker

Although the CSA does not directly provide the symbolic execution, it holds the symbolic values in its inner components. As the CSA carries out the static analysis, it also holds the results of applying SMT solver to the symbolic formulas composing from these symbolic values. The CSA also uses the CFG representation of the program flow, so the symbolic formulas and its expression in numbers can be acquired from the nodes where the CFG is splitting. The number expression are represented by ranges, i.e. every symbolic value should have its range, where is saved and which value should this variable have in the current path.

We applied the CSA to our real-world software and tried to use it for generation of test inputs. To create the test inputs, we implemented a checker, which processed the program after the analysis was done, because we need to work with the complete control flow graph (CFG). We encountered several issues, when trying to do it. First one was caused by the lack of information about how to "hack" the CSA to get the needed information. It was very time consuming and we had to use the trial and error approach. Another problem was the current constraint solver neither did support the bitwise operations nor the floating point numbers. Because the solver is implemented deeply in the code, it is not a simple task to change it for another one. Also the symbolic expressions contain only one symbolic value at the same time, so it is necessary to collect the symbolic values through the CFG. The CSA also enables to work with only one source file at the same time, so to process more files, an extern script should be used. On the other hand, the CSA does not demand the source code to be in the compiled state, and consequently, the CSA can be applied on most software without demanding environment setting.

5.2.4.2 Bugst

The Bugst [128] is an open-source project developed by the team from the Masaryk University in Brno. It is a collection of libraries and utilities aimed to make the developing of experimental tools for the program analysis simpler. The whole project consists of several smaller tools, where each one performs specific function. For example, the Symtex is a library enabling symbolic execution or the Lonka allows user to view the CFG of the analyzed program. The whole Bugst project is written in the C++ and aims to the analysis of the C/C++ code. The ideas implemented in some of its libraries were presented before [107], [129], [130], [131], [132].

To make the Bugst work, we spend a long time in the environment preparation. Finally, we decided to write our own instructions manual [133] for the possible future experimenters, because we wanted to help them from being stuck even before working with the tools. The problems with environment preparation arose because of the lack of detailed documentation and also because the Bugst needed many external libraries. Another cause for difficult preparation is that the Bugst uses the LLVM infrastructure but is written in the form of the Visual Studio project – firstly, one needs to get ready the LLVM infrastructure for the Visual Studio and after that start to build the Bugst. On the contrary, the possibility to use the Bugst in the Visual Studio has its benefits in a good and friendly debugger.

We tried to apply the Bugst tool Rudla to our real-world software. The Rudla is a uniting base for most of the Bugst libraries responsible for the symbolic execution. For example, via Rudla tool, one can start the classic symbolic execution of a program or the compact symbolic execution [129]. To use the Rudla tool correctly, the source code transformations are needed. Firstly, the code has to be compiled via the Clang compiler into the intermediate form of LLVM. After that, the Bugst tools

llvm2cellvm and celllvm2lonka have to be used to convert the LLVM intermediate form into the inner Bugst's simplified form. Right here we encountered first problems, because the simplified form of the Bugst does not support many C++ features. For example, the "anonymous namespaces not supported" exception was thrown very often, even on files, which did not contain any anonymous namespace. Another issue with Rudla occurs during processing more input files. The translator into the intermediate Bugst form is able to work only with one file per transformation and the source code has to contain the Main function. If the input files do not contain it, the transformation will not be successful. On the contrary, when the conversion runs correctly, with the Rudla tool, one can even debug the symbolic execution and collect interesting information about the program flow.

5.2.4.3 KLEE

KLEE [106] is the only one tool we compare here, which can directly generate tests for given input program via the symbolic execution. The KLEE is also the only tool which was already successfully used for the test inputs generation for the C++ code (in the tool KLOVER [112]). Unfortunately, authors of the KLOVER did not make the software open-source and it is not possible to acquire it even for evaluation purposes. The popularity of the KLEE is boosted by the good on-line documentation with easy-to-understand examples, even with one unconventional, where the symbolic execution helps to solve the maze problem [135]. The above declared popularity is also confirmed by recent (August 2014) 43 forks, 89 stars and 27 watches of the source code in the repository [136] and also by many publications and tools, which basis is the KLEE tool [137].

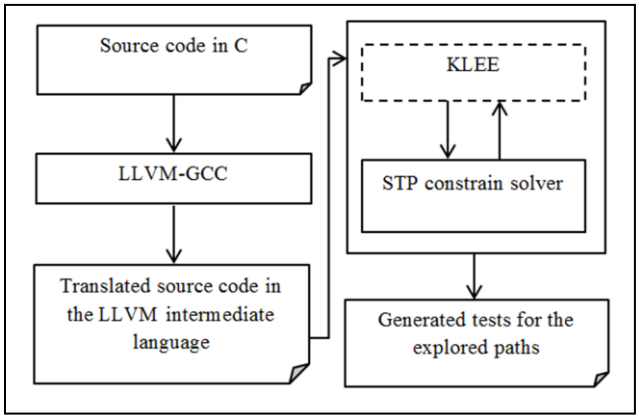

Figure 16. The KLEE work-flow

The base for the KLEE tool was the tool EXE [109]. Unlike the EXE, the KLEE is built over the LLVM infrastructure, can handle the interaction with system environment and brings many other optimizations. The KLEE proved its usefulness many times, for example, it was able to beat in coverage the manual testing of the CoreUtils. The KLEE needed only 89 hours to generate tests and even had over 16.8 % better coverage [106]. Because of its usefulness, KLEE became open-source in 2009 and many users joined the KLEE community. The KLEE has also been extended by several research groups [134].

We tried to apply the KLEE to our real-world software, because it should provide at least limited support for the C++ code [138]. Regrettably, we got stuck even before examining first source file of our code. The KLEE needs to have the analyzed sources compiled into the LLVM intermediate language via the llvm-gcc compiler. The llvm-gcc compiler can be used as substitution for the Clang.

According to the on-line manual, the KLEE uses the llvm-gcc in the version 4.2 [139]. When we tried to compile our code with this compiler, we got many error messages about casting, type sizes etc. and the compilation failed. We found out the problem could lie in the old version of the llvm-gcc, so we installed a new one. We applied the new version to our code and the compilation was done without problems. Unfortunately, the KLEE declined to process compiled sources from the newer compiler version with an error message. We at least tried the examples and found out that the KLEE is quite useful, but at the same time, for our software, we will need to wait for a new version of KLEE with support for new compilers version.

5.2.4.4 Comparison Results

The survey about tools enabling symbolic execution of the C++ code was performed in previous part. Because we have not found any available tool directly allowing the symbolic execution of the C++ code, we tried to "hack" some tools, which partially provide this functionality. We took three tools connected with LLVM for comparison – the Clang Static Analyzer, a static analyzing tool which uses a symbolic execution during the process of a code analysis; the Bugst library, an experimental set of tools for facilitating a creation of code analyzers; and finally the KLEE, which is a tool greatly known for generating tests via symbolic execution for the C code with partial support for C++. In the Table 1, the results are summarized.

	Clang Static Analyzer	Bugst	KLEE
Language	C++	C++	C++
Mainly supported OSs	Linux, MacOS	Windows	Linux
Good to know	Clang	LLVM	LLVM
Pros	Full support for C++, popularity, alive	Many different tools	Aimed at test generation, popularity, alive
Cons	Does not generate test inputs, needs hacks to access to the symbolic values	Does not generate test inputs, does not support all C++ features, non-known	We did not manage to put it into operation with our C++ code
Documentation quality	Good	Poor	Great

Table 12. Tools comparison results.

As we can see in the Table 12, every tool has its cons and pros. It is regrettable that there is no free tool, which would enable the symbolic execution of the C++ code. Instead, one needs to bend the available tools to work for it. We are hesitating which tool should be recommended as a base for building a symbolic executor for the C++ for the purpose of test generation. All three examples have

its benefits, but it will cost a lot of effort to enhance any of them to generate tests for the C++ code. Finally, we let the reader to decide, which tool would suit better for him, because everybody has its own criteria. As we wrote before, it will cost many hours of coding to add the missing functionality. Consequently, it is hard to conclude with a clear statement about which tool should be chosen.

6 Conclusion

In the first part of the thesis, the general term quality was introduced and several definitions of quality were presented. With sufficient background from the general definitions of quality, the software quality was defined. Three well-known definitions of software quality were shown and their differences were discussed. Afterwards, the idea of quality models as means for evaluation of software quality was introduced. Three important software quality models were covered and their inner structure was described thoroughly. In the last part of the first section, the software metrics were announced as an important measurement tool in the software quality models. The first section covers broad part of software quality in the common context and therefore can be handy as a source of information for anyone interested in the area of software quality.

The second part of the thesis covers the topic of safety critical systems. After introduction into the topic and definition of safety critical systems, three methods used in recently developed safety critical systems were shown. All three methods were described in detail, which can give abundant information about how the safety critical software systems are developed. Finally, comparison between the three methods was produced and the result can help possible feature developers of safety critical systems to decide which design implementation to choose.

In the third part of the thesis, the topic of evaluation of quality of safety critical systems was covered. Firstly, the metrics relevant for safety critical systems were selected from two real-world safety critical projects. Three tools enabling measurement of selected metrics were explored and described together with mapping of the selected metrics into the tools API. Later on, sample project was measured by the selected tools and by selected metrics. As a next step, differences in measurements from the three tools were discussed. The mapping and results examination described can help other practitioners in selecting right tool and metrics for quality assessment. Last but not least, the topic of metrics thresholds was discussed in-depth and thresholds for particular safety critical system were derived. Because the metrics were clearly selected, the thresholds can be used by possible future evaluators of another safety critical system. Finally, based on the metrics thresholds gathered, the system quality was evaluated. Owing to the quality evaluation, several parts of the code were recommended for further review.

The fourth part of the thesis aimed at the quality improvement of safety critical systems. The main idea lies in enhancing quality of the systems through improving results of software metrics. To enhance the quality more rapidly, the metrics importance was placed and one particular metric with great importance, test coverage, was chosen. As a next step, methods which can improve the metrics results were discussed. One of the methods – test input generation via symbolic execution – was selected because of its ability to widely enhance the number of tests. The method is deeply described and its advantages and challenges were presented. Afterwards, three tools partially enabling application of the method to real-world safety critical software were introduced and compared thoroughly. This part of the thesis can be beneficial for those, who want to improve quality of not-just safety critical systems, because it indicates the possible way how to do it. The described method of test generation via symbolic execution and discussion of its challenges can also help the future developers to determine critical parts of the implementation.

6.1 Possible Directions of Future Research

In the thesis, various topics connected with quality of safety critical systems were covered and discussed. Because of the complexity of covered topics, there are several possible directions where the future research can lead. Some of them will be discussed below.

The metrics thresholds evaluation in the fourth part of the thesis considered only one particular safety critical system. The future endeavor can be seen in the study of differences between metrics thresholds measured in this thesis and data from a system with at least some degree of similarity. As it has already been said in the section, it is hard to collect data from systems as was the one which was examined because they are usually not publicly available. Nevertheless, it would be interesting to compare the gathered results with measurement of thresholds of some open-source operational systems such as Linux that appear to be the most similar to our software. The research should be done with bearing the domain differences in mind. Another approach for selection of proper thresholds could be mining the version control systems used for the current safety critical system. With sufficient knowledge of the source code from the repository, the thresholds can show how the quality of the system flown in the time.

Several opened questions still remain regarding to the importance of metrics and weights. For example, the test coverage metric was chosen as important by examination of several well-known norms for safety critical systems and its broad usage. As a possible tool for important metrics' selection, their weights should be considered. The research in the area of metrics weights is still ongoing and at the moment, there is no unitary view on weights evaluation. For example, the author of [14] defines wide set of metrics which can be used for measurement and also provides the mapping between software characteristics, sub-characteristics and metrics. The mapping is supplied in the form of on-line compendium [140] accessible for everyone with the Internet access. Authors of the compendium also provide kind of metrics' weight assessment by defining each connection between sub-characteristic and metric with description "related" / "highly related". Even if the weights are not numerical, it can be useful to get impression about how to relate and weight metrics. Another approach can be seen in the work [32] and [33] where the authors use multi-criteria analysis named Analytic Hierarchy Process to derive weights for selected metrics and afterwards, they use them together with clustering algorithm for system quality evaluation. The approach of the authors is interesting, but the paper [32] suffers from the lack of proper description of the method used and computed weights for metrics. The paper [33] contains information about weights and proper description of applied method, but also reveals that subjective expert knowledge is needed to set the initial metrics weights. The effort to put importance into metrics measurement can be seen also in the ISO 9126-1 [13]. The authors provide several examples with weight assigned to software sub-characteristics. Regrettably, the weights examples do not connect metrics with sub-characteristics.

As it was shown in the section about symbolic execution, there are many unsolved challenges the resolving of which can greatly improve the method itself. Several challenges such as path explosion, extern calls or complex constrains were already discussed in the text, but there are also many others, which were not covered to preserve reasonable scope of the text. For example, the floating-point computation is the known challenge which was not discussed in here, but the improvement of which would enhance the test generation widely. Recently, this challenge was targeted for example in [141], where the authors use static analysis of floating point instructions to overcome this issue. Despite preliminary experiments showed the method to be able to alleviate the solving of floating

point constraints, the technique suffers from false positives and there is still room for an improvement.

7 References

[1] Feigenbaum, Armand V. "Total quality-control." Harvard Business Review 34.6 (1956): 93-101.

[2] Crosby, P. B., Quality is free : the art of making quality certain, New York : McGraw-Hill, 1979.

[3] Shewhart, Walter Andrew. Economic control of quality of manufactured product. Vol. 509. ASQ Quality Press, 1931.

[4] Hoyer, Robert W., and Brooke BY Hoyer. "What is quality." Quality Progress 34.7 (2001): 53-62.

[5] ISO 9000.1:1994, Quality Management and Quality Assurance Standards; Part 1: Guidelines for Selection and Use, International Organisation for Standardisation, Geneva.

[6] American Society for Quality, Glossary – Entry: Quality, retrieved 2015-04-22

[7] Biehl R., Six Sigma for Software, IEEE Software, IEEE, Vol. 21, No. 2, 2004, pp. 68-71.

[8] Wikipedia contributors. "Quality (business)." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 7 Apr. 2015. Web. 22 Apr. 2015.

[9] Pressman, Roger S. Software engineering: a practitioner's approach. Palgrave Macmillan, 2005, pp. 199.

[10] ISO. ISO/IEC 14598-1: Software Engineering - Product evaluation – Part 1: General.

[11] IEEE Computer Society. Software Engineering Technical Committee. "IEEE standard glossary of software engineering terminology." Institute of Electrical and Electronics Engineers, 1990.

[12] Wagner, Stefan. Software product quality control. Berlin: Springer, 2013, pp. 22.

[13] ISO. ISO/IEC 9126-1 “Software engineering - Product Quality - Part 1: Quality model”, 2001.

[14] R. Lincke. Validation of a Standard- and Metric-Based Software Quality Model -- Creating the Prerequisites for Experimentation. Licentiate thesis, MSI, Växjö University, Sweden, Apr 2007.

[15] McCall, Jim A., Paul K. Richards, and Gene F. Walters. Factors in software quality. Concepts and definitions of software quality. GENERAL ELECTRIC CO SUNNYVALE CA, 1977.

[16] Boehm, Barry W., John R. Brown, and Hans Kaspar. "Characteristics of software quality." (1978).

[17] Boehm, Barry W., John R. Brown, and Mlity Lipow. "Quantitative evaluation of software quality." Proceedings of the 2nd international conference on Software engineering. IEEE Computer Society Press, 1976.

[18] Berander, Patrik, et al. "Software quality attributes and trade-offs." Blekinge Institute of Technology (2005).

[19] ISO. ISO/IEC 9126-2 “Software engineering - Product Quality - Part 2: External metrics”, 2003.

[20] ISO. ISO/IEC 9126-3 “Software engineering - Product Quality - Part 3: Internal metrics”, 2003.

[21] ISO. ISO/IEC 9126-4, “Software Engineering – Product Quality – Part 4: Quality in Use Metrics”, 2004.

[22] ISO. ISO/IEC 25000:2011 “Systems and software engineering, Systems and software Quality Requirements and Evaluation (SQuaRE), System and software quality models”, 2011.

[23] Jamwal, Deepshikha. "Analysis of software quality models for organizations." International Journal of Latest Trends in Computing 1.2 (2010).

[24] B. Al-Badareen, M. H. Selamat, M. A. Jabar, H. Din and S. Turarv, “Software Quality Model: A Comparative Study,” Springer ICSECS’11, Page No.: 46 - 55, 2011.

[25] Lincke, R.; Gutzmann, T.; Löwe, W., "Software Quality Prediction Models Compared," Quality Software (QSIC), 2010 10th International Conference on , vol., no., pp.82,91, 14-15 July 2010.

[26] Kaur, Sarabjit, Satwinder Singh, and Harshpreet Kaur. "A Quantitative Investigation Of Software Metrics Threshold Values At Acceptable Risk Level." International Journal of Engineering Research and Technology. Vol. 2. No. 3 (March-2013). ESRSA Publications, 2013.

[27] Ferreira, Kecia AM, et al. "Identifying thresholds for object-oriented software metrics." Journal of Systems and Software 85.2 (2012): 244-257.

[28] Chidamber, Shyam R., and Chris F. Kemerer. "A metrics suite for object oriented design." Software Engineering, IEEE Transactions on 20.6 (1994): 476-493.

[29] Li, Wei, and Sallie Henry. "Object-oriented metrics that predict maintainability." Journal of systems and software 23.2 (1993): 111-122.

[30] Bieman, James M., and Byung-Kyoo Kang. "Cohesion and reuse in an object-oriented system." ACM SIGSOFT Software Engineering Notes. Vol. 20. No. SI. ACM, 1995.

[31] Hitz, Martin, and Behzad Montazeri. Measuring coupling and cohesion in object-oriented systems., pp. 24, 25, 274, 279, 1995.

[32] Antonellis, P., et al. "A Data Mining Methodology for Evaluating Maintainability according to ISO/IEC-9126 Software Engineering-Product Quality Standard." Special Session on System Quality and MaintainabilitySQM2007 (2007).

[33] Kanellopoulos, Yiannis, et al. "Code quality evaluation methodology using the ISO/IEC 9126 standard." arXiv preprint arXiv:1007.5117 (2010).

[34] Knight, John C. "Safety critical systems: challenges and directions." Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on. IEEE, 2002.

[35] "The European Rail Traffic Management System." ERTMS. Web. 25 Apr. 2015. <http://www.ertms.net/>.

[36] CELENEC, EN 50128 Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems, 2nd ed., 2011.

[37] Geffroy, Jean-Claude, and Gilles Motet. Design of dependable computing systems. Kluwer Academic Publishers, 2002.

[38] Chalin, Patrice. "Are practitioners writing contracts?." Rigorous Development of Complex Fault-Tolerant Systems. Springer Berlin Heidelberg, 2006. 100-113.

[39] B. Meyer, Object-Oriented Software Construction, 1st ed., Prentice-Hall, Inc., 1998, pp. 331-410.

[40] Eris, Oytun, et al., “Application of functional safety on railways part III: Developent of a main controller,” Control Conference (ASCC), 2011 8th Asian. IEEE, 2011.

[41] Durmus, M. S., U. Yildirim, and M. T. Soylemez. "Application of functional safety on railways part I: Modelling & design." Control Conference (ASCC), 2011 8th Asian. IEEE, 2011.

[42] Yildirim, Ugur, M. S. Durmus, and M. T. Soylemez. "Application of functional safety on railways part II: Software development." Control Conference (ASCC), 2011 8th Asian. IEEE, 2011.

[43] Durmus, M. S., et al. "A new voting strategy in Diverse programming for railway interlocking systems." Transportation, Mechanical, and Electrical Engineering (TMEE), 2011 International Conference on. IEEE, 2011.

[44] IEC 61508 Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related System, 1998.

[45] Eris, Oytun, et al. “NVP for Railway Interlocking Systems: Synchronization and Voting Strategy,” 13th IFAC Symposium on Control in Transportation Systems, CTS 2012, Sofia, pp. 12-14, 2012.

[46] Eris, Oytun, et al. "Comparison of the Parallel and Serial Architectures for N-Version Programming As Applied to Railway Interlocking Systems." Control and Automation Theory for Transportation Applications. Vol. 1. No. 1. 2013.

[47] Pullum, Laura L. Software fault tolerance techniques and implementation. Artech House, pp. 270-273, 2001.

[48] D.W. Carr, R. Ruelas, J.F. Gutierrez-Ramirez, H. Salcedo-Becerra, “An Open On-Board CBTC Controller Based on NVP,” Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce, vol.1, pp.834-839, 2005.

[49] Rosenberg, Linda H., and Lawrence E. Hyatt. "Software quality metrics for object-oriented environments." Crosstalk journal 10.4 (1997).

[50] Reinhardt, Derek W. "Use of the C++ Programming Language in Safety Critical Systems." Masters of Science in Safety-Critical Systems Engineering at the Department of Computer Science the University of York (2004).

[51] "DigitalInBlue/Celero." GitHub. Web. 23 Apr. 2015. <https://github.com/DigitalInBlue/Celero>.

[52] "Understand Your Code | SciTools.com." SciTools.com. Web. 23 Apr. 2015. <https://scitools.com>.

[53] "Static Analysis and Coding Standards Compliance | PRQA." PRQA Programming Research RSS. Web. 23 Apr. 2015. <http://www.programmingresearch.com/>.

[54] "CppDepend :: Achieve Higher C/C Code Quality." Improve Your C/C Code Quality with CppDepend. Web. 23 Apr. 2015. <http://www.cppdepend.com/>.

[55] Lincke, Rüdiger, Jonas Lundberg, and Welf Löwe. "Comparing software metrics tools." Proceedings of the 2008 international symposium on Software testing and analysis. ACM, 2008.

[56] Alves, Tiago L., Christiaan Ypma, and Joost Visser. "Deriving metric thresholds from benchmark data." Software Maintenance (ICSM), 2010 IEEE International Conference on. IEEE, 2010.

[57] Herbold, Steffen, Jens Grabowski, and Stephan Waack. "Calculation and optimization of thresholds for sets of software metrics." Empirical Software Engineering 16.6 (2011): 812-841.

[58] Marinescu, Radu, and Michelle Lanza. "Object-oriented metrics in practice." (2006).

[59] De Souza, Lucas Batista Leite, and Marcelo De Almeida Maia. "Do software categories impact coupling metrics?." Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press, 2013.

[60] "Support | SciTools.com." SciTools.com. Web. 23 Apr. 2015. <https://scitools.com/support/metrics_list/>.

[61] Shatnawi, Raed, and Qutaibah Althebyan. "An Empirical Study of the Effect of Power Law Distribution on the Interpretation of OO Metrics." ISRN Software Engineering 2013 (2013).

[62] Tamai, Tetsuo, and Takako Nakatani. "Analysis of software evolution processes using statistical distribution models." Proceedings of the International Workshop on Principles of Software Evolution. ACM, 2002.

[63] Katz, R. W. "Do weather or climate variables and their impacts have heavy-tailed distributions." Proc. of 16th Conference on Probability and Statistics in the Atmospheric Sciences, American Meteorological Society. 2002.

[64] Gardiner, Joseph C. "Modeling heavy-tailed distributions in healthcare utilization by parametric and Bayesian methods." SAS Global Forum. 2012.

[65] Oliveira, Paloma, et al. "Metrics-based Detection of Similar Software." INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING. Proceedings. sn, 2013.

[66] Abran, Alain. Software metrics and software metrology. John Wiley & Sons, 2010, page 210.

[67] Mindance.Testing Metrics Whitepaper. http://www.mindlance.com/documents/test_management/testing_metrics.pdf

[68] Infosys. Metrics Model Whitepaper. http://www.infosys.com/engineering-services/white-papers/Documents/comprehensive-metrics-model.pdf

[69] NASA Software Safety Guidebook. MIL-STD-882C, http://www.hq.nasa.gov/office/codeq/doctree/871913.pdf

[70] RTCA Inc., "Software Considerations in Airborne Systems and Equipment Certification," Washington, D.C. RTCA/DO-178B, 1992.

[71] Mathur, Aditya P. Foundations of Software Testing, 2/e. Pearson Education India, 2008.

[72] Tassey, Gregory. "The economic impacts of inadequate infrastructure for software testing." National Institute of Standards and Technology, RTI Project 7007.011 (2002).

[73] "Vulnerability Note VU#720951." - OpenSSL TLS Heartbeat Extension Read Overflow Discloses Sensitive Information. Web. 23 Apr. 2015. <http://www.kb.cert.org/vuls/id/720951>.

[74] "How to Prevent the next Heartbleed." How to Prevent the next Heartbleed. Web. 23 Apr. 2015. <http://www.dwheeler.com/essays/heartbleed.html>.

[75] Ammann, Paul, and Jeff Offutt. Introduction to software testing. Cambridge University Press, 2008.

[76] Kobayashi, Noritaka. "Design and evaluation of automatic test generation strategies for functional testing of software." Osaka, Japan, Osaka Univ (2002).

[77] "PřeVODNÍK." PřeVODNÍK. Web. 23 Apr. 2015. <http://oks.kiv.zcu.cz/Prevodnik>.

[78] Motor Industry Research Association, “MISRAC++:2008- Guidelines for the use of the C++ language in critical systems, 2008

[79] M. Young, The Technical Writer’s Handbook. Mill Valley, CA: University Science, 1989.

[80] Ferguson, Roger, and Bogdan Korel. "The chaining approach for software test data generation." ACM Transactions on Software Engineering and Methodology (TOSEM) 5.1 (1996): 63-86.

[81] Grieskamp, Wolfgang, et al. "Action machines-towards a framework for model composition, exploration and conformance testing based on symbolic computation." Quality Software, 2005.(QSIC 2005). Fifth International Conference on. IEEE, 2005.

[82] Visser, Willem, Corina S. Păsăreanu, and Radek Pelánek. "Test input generation for java containers using state matching." Proceedings of the 2006 international symposium on Software testing and analysis. ACM, 2006.

[83] Anand, Saswat, et al. "An orchestrated survey of methodologies for automated software test case generation." Journal of Systems and Software 86.8 (2013): 1978-2001.

[84] Pacheco, Carlos, and Michael D. Ernst. "Randoop: feedback-directed random testing for Java." Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion. ACM, 2007.

[85] Pacheco, Carlos, et al. "Feedback-directed random test generation." Software Engineering, 2007. ICSE 2007. 29th International Conference on. IEEE, 2007.

[86] Kuhn, D. Richard, Dolores R. Wallace, and Jr AM Gallo. "Software fault interactions and implications for software testing." Software Engineering, IEEE Transactions on 30.6 (2004): 418-421.

[87] Kuhn, D. Richard, Raghu N. Kacker, and Yu Lei. "Practical combinatorial testing." NIST Special Publication 800.142 (2010): 142.

[88] "Automated Combinatorial Testing for Software (ACTS)." NIST Computer Security Division Web. 23 Apr. 2015. <http://csrc.nist.gov/groups/SNS/acts/index.html>

[89] "Hexawise – Pairwise Testing Made Easy." Hexawise – Pairwise Testing Made Easy. Web. 23 Apr. 2015. <https://hexawise.com/>.

[90] "Tutorials Point - Simply Easy Learning." All-Pairs Testing. Web. 23 Apr. 2015. <http://www.tutorialspoint.com/software_testing_dictionary/all_pairs_testing.htm>.

[91] Pretschner, Alexander. "Model-based testing." Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on. IEEE, 2005.

[92] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT\—a formal system for testing and debugging programs by symbolic execution. In Proceedings of the international conference on Reliable software. ACM, New York, NY, USA, 234-245.

[93] Osterweil, Leon J., and Lloyd D. Fosdick. "Program testing techniques using simulated execution." ACM SIGSIM Simulation Digest. Vol. 7. No. 4. IEEE Press, 1976, pages 171–177.

[94] William E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. IEEE Transactions on Software Engineering, 3(4):266–278, 1977.

[95] King, James C. "Symbolic execution and program testing." Communications of the ACM 19.7 (1976): 385-394.

[96] Ganesh, Vijay, and David L. Dill. "A decision procedure for bit-vectors and arrays." Computer Aided Verification. Springer Berlin Heidelberg, 2007.

[97] De Moura, Leonardo, and Nikolaj Bjørner. "Z3: An efficient SMT solver." Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2008. 337-340.

[98] Dutertre, Bruno, and Leonardo De Moura. "A fast linear-arithmetic solver for DPLL (T)." Computer Aided Verification. Springer Berlin Heidelberg, 2006.

[99] "Linux Hardware Reviews, Open-Source Benchmarks & Linux Performance - Phoronix." Linux Hardware Reviews, Open-Source Benchmarks & Linux Performance - Phoronix. Web. 23 Apr. 2015. <http://www.phoronix.com/scan.php?page=news_item&px=MTE4NjU>.

[100] Sen, Koushik, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. Vol. 30. No. 5. ACM, 2005.

[101] Godefroid, Patrice, Michael Y. Levin, and David Molnar. "SAGE: whitebox fuzzing for security testing." Queue 10.1 (2012): 20.

[102] Godefroid, Patrice, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." ACM Sigplan Notices. Vol. 40. No. 6. ACM, 2005.

[103] Joshi, Pallavi, Koushik Sen, and Mark Shlimovich. "Predictive testing: amplifying the effectiveness of software testing." The 6th Joint Meeting on European software engineering

conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers. ACM, 2007.

[104] Pasareanu, Corina S., and Willem Visser. "A survey of new trends in symbolic execution for software testing and analysis." International journal on software tools for technology transfer 11.4 (2009): 339-353.

[105] Godefroid, Patrice, Michael Y. Levin, and David A. Molnar. "Automated Whitebox Fuzz Testing." NDSS. Vol. 8. 2008.

[106] Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." OSDI. Vol. 8. 2008.

[107] Jan Obdržálek, Marek Trtík. Efficient Loop Navigation for Symbolic Execution. ATVA 2011, Springer, p. 453-462.

[108] Burnim, Jacob, and Koushik Sen. "Heuristics for scalable dynamic test generation." Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering. IEEE Computer Society, 2008.

[109] Cadar, Cristian, et al. "EXE: automatically generating inputs of death." ACM Transactions on Information and System Security (TISSEC) 12.2 (2008): 10.

[110] Willink, Edward D.: "Meta-compilation for C++." PhD Thesis, Computer Science Research Group, University of Surrey, January 2000

[111] Garg, Pranav, et al. "Feedback-directed unit test generation for C/C++ using concolic execution." Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 2013.

[112] Li, Guodong, Indradeep Ghosh, and Sreeranga P. Rajan. "KLOVER: A symbolic execution and automatic test generation tool for C++ programs." Computer Aided Verification. Springer Berlin Heidelberg, 2011.

[113] "The LLVM Compiler Infrastructure Project." The LLVM Compiler Infrastructure Project. Web. 23 Apr. 2015. <http://llvm.org/>.

[114] "UClibc." C Library. Web. 23 Apr. 2015. <http://cxx.uclibc.org/>.

[115] Garg, Pranav, et al. “Feedback-directed random class unit test generation using symbolic execution.” Patent US20130091495. 11 April 2013.

[116] Necula, George C., et al. "CIL: Intermediate language and tools for analysis and transformation of C programs." Compiler Construction. Springer Berlin Heidelberg, 2002.

[117] "Ahorn/native-symbolic-execution-clang." GitHub. Web. 23 Apr. 2015. <https://github.com/ahorn/native-symbolic-execution-clang>.

[118] "STANCE: A Source Code Analysis Toolbox for Software Security AssuraNCE - A Source Code Analysis Toolbox for Software Security AssuraNCE." STANCE: A Source Code Analysis Toolbox for Software Security AssuraNCE - A Source Code Analysis Toolbox for Software Security AssuraNCE. Web. 23 Apr. 2015. <http://stance-project.eu/>.

[119] "Available Research Tools - Asergrp." Available Research Tools - Asergrp. Web. 23 Apr. 2015. <https://sites.google.com/site/asergrp/tools>.

[120] "Code-based Test Generation." Overview and Tools. Web. 23 Apr. 2015. <http://home.mit.bme.hu/~micskeiz/pages/code_based_test_generation.html>.

[121] Chen, Ting, et al. "State of the art: Dynamic symbolic execution for automated test generation." Future Generation Computer Systems 29.7 (2013): 1758-1773.

[122] Qu, Xiao, and Brian Robinson. "A case study of concolic testing tools and their limitations." Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on. IEEE, 2011.

[123] "Clang: A C Language Family Frontend for LLVM." "clang" C Language Family Frontend for LLVM. Web. 23 Apr. 2015. <http://clang.llvm.org/>.

[124] "GCC 4.9 vs. LLVM Clang 3.5 Linux Compiler Benchmarks." GCC 4.9 Vs. LLVM Clang 3.5 Linux Compiler Benchmarks Performance. Web. 23 Apr. 2015. <http://openbenchmarking.org/result/1404144-KH-CLANG359076>.

[125] "Clang Static Analyzer." Clang Static Analyzer. Web. 23 Apr. 2015. <http://clang-analyzer.llvm.org/index.html>.

[126] "Using Static Analysis and Clang To Find Heartbleed." And You Will Know Us by the Trail of Bits. 27 Apr. 2014. Web. 23 Apr. 2015. <http://blog.trailofbits.com/2014/04/27/using-static-analysis-and-clang-to-find-heartbleed/>.

[127] "Rettichschnidi/clang-misracpp2008." GitHub. Web. 23 Apr. 2015. <https://github.com/rettichschnidi/clang-misracpp2008>.

[128] "Bugst." SourceForge. Web. 23 Apr. 2015. <http://sourceforge.net/projects/bugst/>.

[129] Slaby, Jiri, Jan Strejček, and Marek Trtík. "Compact symbolic execution."Automated Technology for Verification and Analysis. Springer International Publishing, 2013. 193-207.

[130] Slaby, Jiri, Jan Strejček, and Marek Trtík. "Symbiotic: synergy of instrumentation, slicing, and symbolic execution." Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2013. 630-632.

[131] Strejček, Jan, and Marek Trtík. "Abstracting path conditions." Proceedings of the 2012 International Symposium on Software Testing and Analysis. ACM, 2012.

[132] Slabý, Jiří, Jan Strejček, and Marek Trtík. "Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution." Formal Methods for Industrial Critical Systems. Springer Berlin Heidelberg, 2012. 207-221.

[133] "Stepan´s Digital Notepad." : Detailed Manual How to Install The Bugst. Web. 2 May 2015. <https://stepnpad.blogspot.com/2014/08/detailed-manual-how-to-install-bugst.html>.

[134] Cadar, Cristian, and Koushik Sen. "Symbolic execution for software testing: three decades later." Communications of the ACM 56.2 (2013): 82-90.

[135] "The Symbolic Maze!" Feliams Blog. 7 Oct. 2010. Web. 23 Apr. 2015. <http://feliam.wordpress.com/2010/10/07/the-symbolic-maze/>.

[136] "Klee/klee." GitHub. Web. 23 Apr. 2015. <https://github.com/klee/klee>.

[137] Web. 23 Apr. 2015. <http://klee.github.io/klee/Publications.html>.

[138] "[klee-dev] Checking Llvm Build Mode... Configure: Error: Invalid Build Mode:." [klee-dev] Checking Llvm Build Mode... Configure: Error: Invalid Build Mode:. Web. 23 Apr. 2015. <http://mailman.ic.ac.uk/pipermail/klee-dev/2013-January/000031.html>.

[139] "Getting Started · KLEE." Getting Started · KLEE. Web. 23 Apr. 2014. <http://klee.github.io/klee/GetStarted.html>.

[140] "Compendium of Software Quality Standards and Metrics - Version 1.0." Compendium of Software Quality Standards and Metrics - Version 1.0. Web. 23 Apr. 2015. <http://www.arisa.se/compendium/>.

[141] Godefroid, Patrice, and Johannes Kinder. “Proving memory safety of floating-point computations by combining static and dynamic program analysis” in: Proceedings of the International Symposium on Software Testing and Analysis, 2010, pp. 1–11.