A Weighted Grid for Measuring Program Robustness

Mohammad Mahmoud Aref
ABDALLAH

School of Engineering and Computing Sciences

Durham University

A thesis submitted for the degree of
Doctor of Philosophy

2012

Abstract

Robustness is a key issue for all the programs, especially safety critical ones. In the literature, Program Robustness is defined as “the degree to which a system or component can function correctly in the presence of invalid input or stressful environment” (IEEE 1990). Robustness measurement is the value that reflects the Robustness Degree of the program.

In this thesis, a new Robustness measurement technique; the Robustness Grid, is introduced. The Robustness Grid measures the Robustness Degree for programs, C programs in this instance, using a relative scale. It allows programmers to find the program’s vulnerable points, repair them, and avoid similar mistakes in the future.

The Robustness Grid is a table that contains Language rules, which is classified into categories with respect to the program’s function names, and calculates the robustness degree. The Motor Industry Software Reliability Association (MISRA) C language rules with the Clause Program Slicing technique will be the basis for the robustness measurement mechanism.

In the Robustness Grid, for every MISRA rule, a score will be given to a function every time it satisfies or violates a rule. Furthermore, Clause program slicing will be used to weight every MISRA rule to illustrate its importance in the program. The Robustness Grid shows how much each part of the program is robust and effective, and assists developers to measure and evaluate the robustness degree for each part of a program.

Overall, the Robustness Grid is a new technique that measures the robustness of C programs using MISRA C rules and Clause program slicing. The Robustness Grid shows the program robustness degree and the importance of each part of the program. An evaluation of the Robustness Grid is performed to show that it offers new measurements that were not provided before.

Dedication

I dedicate this thesis to

my late father who had a dream that one day I will finish my PhD, and I did it for him.

My mum who without her prayers and care I will not be able to finish this research.

Nuha, who support me all through the research journey.

and my brothers, sisters, nephews, nieces, and friends...

Acknowledgement

First of all, all praise and thanks to Allah for the success in this thesis in particular and in my life in general.

My deep thanks to my supervisors Professor Malcolm Munro and Dr. Keith Gallagher for their invaluable advices and guidance. Professor Malcolm and Dr. Keith gave me all benefits from their wide experience. My thesis would not be completed and success without their great supervision.

My great thanks to Dr. Ayman and Dr. Aref who gave me the full financial support and their support gave me the ability to do this research.

My sincere thanks to Mr Chris Tapp, LDRA Company, Mr James Widman, and Gimple Software Company for their help evaluate the proposed technique in this research.

I would like to thank all my friends and colleagues, specially Ahmad, Amir, Maria, Khalid, Rushdie, Wei, and all my colleagues.

Copyright

Declaration

The material contained within this thesis has not previously been submitted for a degree at Durham University or any other university. The Following publications were produced during the course of this thesis:

- Abdallah, M., M. Munro, and K. Gallagher. “Certifying Software Robustness Using Program Slicing”, *In Proceeding of the IEEE International Conference on Software Maintenance (ICSM)*. Timisoara, Romania. 2010, pp.1-2.

- Abdallah, M., M. Munro, and K. Gallagher. “A Static Robustness Grid Using MISRA C2 Language Rules”, *In Proceeding of the Sixth International Conference on Software Engineering Advances (ICSEA 2011)*. Barcelona, Spain. 2011, pp.65-69.

Contents

Chapter Three: Robustness Grid

Chapter Four: Implementation

Chapter Five: Results

Chapter Six: Evaluation

Chapter Seven: Conlusions and Future Work

List of Tables

Chapter Two

Chapter Three

Chapter Four

Chapter Five

Chapter Six

List of Figures

Chapter Two

Chapter Three

Chapter Four

Chapter Five

Chapter Six

Acronyms

FCD: Function Category Degree.

FCSD: Function Category Satisfied Degree.

FCVD: Function Category Violated Degree.

PCD: Program Category Degree.

PCSD: Program Category Satisfied Degree.

PCVD: Program Category Violated Degree.

AC: Accumulative Categories.

FAC: Function Accumulative Categories.

FSAC: Function Satisfied Accumulative Categories.

FVAC: Function Violated Accumulative Categories

WCFD: Whole Category Function Degree.

WCFSD: Whole Category Function Satisfied Degree.

WCFVD: Whole Category Function Violated Degree.

WPD: Whole Program Degree.

WPSD: Whole Program Satisfied Degree.

WPVD: Whole Program Violated Degree.

WPW: Whole Program Weight.

Chapter One

Introduction

1.1 Research Overview

Software Engineering may be defined as: "*The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software*" (IEEE 1990). The definition shows the process of the software lifecycle.

The software developer's target is to write a program that meets all the specifications and never fails. Software Testing and Verification checks whether the software was correctly built and developed, and meets the specifications. There are different aspects that are tested and verified depending on the Software and its specifications.

The general terms that developers target form Software Dependability. It means "*the ability to deliver service that can justifiably be trusted*" (Avizienis, Laprie et al. 2004).

The primary aspects of Dependability include: availability, reliability, safety, maintainability and integrity. Besides these primary aspects, there is a secondary level of Dependability attributes and one of these attributes is Software Robustness (Avizienis, Laprie et al. 2004). Software Robustness means that the software is able to operate under stress or tolerate unpredictable or invalid input. It is strongly related to the Software Correctness term, where robust software can function correctly even in unusual situations.

Different techniques were used to enhance and measure Software Robustness. Fault detection and prevention are two ideas that have been used in Fault tolerance, Redundancy, and Agents systems. These techniques are applied to develop fault-free programs.

A program robustness measurement is the assessment of how a program can face different problems. In this research, a new technique is introduced to measure the Program Robustness for C language programs. The new technique is called the Robustness Grid. In the Robustness Grid, C language programs robustness is measured using the MISRA C2 language rules. The Robustness Grid is a table that shows the Robustness measurement results for a C program from different points of view.

The Clause Slicing technique was introduced to weight the rules and the code showing different levels of importance. Clause Slicing is a Static Slicing technique, but the Slicing criteria use a piece of code; the Clause.

The Robustness Grid aims to facilitate the program maintenance process. It shows the weak points and their importance in the program, so the maintainer will have enough details to make improvements. Both MISRA C2 rules and Clause Slicing are used to build the Robustness Grid.

The Robustness Grid is compared with other techniques that use MISRA C2 rules. The evaluation of the Robustness Grid will show that it offers new measurements that were not provided before.

The motivation for this research is that all programs should be robust to execute their tasks without any trouble. However, most of the programs are not fully robust, and still have some weak points that could cause some errors. Though, there are many techniques that test the program robustness still there are very few that measure it, and help developer to increase the robustness of a program. In this research, a robustness measurement is introduced that shows the developers exactly where are the weak robustness points and to which level they are important and need to be fixed. The measurement results are shown in a numerical table called Robustness Grid which is a new way of presenting measurement results.

The Robustness Grid can be applied in the real life, where it gives the developers and maintainers an indication to the parts of the program that need to be repaired to improve the Robustness Degree of a C program.

In the Robustness Grid, the program robustness measurement results can be presented in different levels of details, and that make it suitable to be presented and explained to all levels of developers team hierarchy.

The main contributions in this thesis are:

1- A new program slicing technique called Clause Slicing is introduced and defined. It enables the program code lines and statements to be analysed fully and show their influence on the program.

2- Using program Clause Slicing to measure the robustness of a program is a new technique. The Clause Slicing is used to give different weights for the program clauses and rules that measure the Robustness.

3- The Robustness Grid is the main contribution and is a table that shows the robustness measurement results in a numerical presentation form.

1.2 Criteria for Success

In this thesis, the criteria for success are the development, implementation, analysis and evaluation of the new robustness measurement Grid. The criteria for success are set as follows:

1- **Develop a measurement for assessing the Robustness of C programs.**

   C Program Robustness will be measured against a set of rules from a language standard. Slicing is also used in weighting these language rules. All measurements will be presented in the Robustness Grid.

2- **Develop a Grid that incorporates the robustness measurement.**

   The Robustness Grid is a table that shows the Robustness measurement. The Robustness will be measured for each function, and for the entire program. The measurement will be presented in relative and absolute numbers. These numbers give an indication of the Robustness state of the program and its functions and an indication of the effect that each piece of code has in the program.

3- **Empirically evaluate the Grid.**

   Robustness will be evaluated by assessing robustness will be evaluated and assessed with a major case study.

4- **Compare the results against other related studies.**

   The Robustness Grid will be compared with other robustness measurement techniques. The evaluation will show the contributions of the Robustness Grid and will enable the evaluation of its limitations. It will also show whether the Robustness Grid provides an accurate Robustness measurement.

5- **Develop a proof of concept of implementation.**

A prototype proof of concept will be presented in order to demonstrate that the new approach is implemented and viable.

1.3 Thesis Outline

The thesis is divided into three main parts; the background, the proposed Grid, and the evaluation. The background consists of Chapter 1 and the Literature Review of Program Robustness and Program Slicing in Chapter 2. The proposed Grid is described in Chapter 3. Implementation of the Robustness Grid is described in Chapter 4. The description involves a major case study in Chapter 5, and the analysis of the Grid will be discussed in Chapter 6. A conclusion and future research directions are presented in Chapter 7.

Chapter 2 reviews basic knowledge about Program Robustness and Program Slicing. The chapter provides the definitions of Robustness, techniques, and tools that have been used in previous studies to measure program robustness. It also explores different program slicing techniques, applications, and tools that provide program analysis.

In Chapter 3, the new robustness measurement Grid, called the Robustness Grid, is introduced. The Robustness Grid uses a new static slicing technique called Clause Slicing and C language rules called MISRA C2 rules to measure C programs robustness.

Chapter 4 shows the implementation road map of the Robustness Grid. It shows the existing tools used to apply the Robustness Grid.

Chapter 5 will present a case study that shows how the Robustness Grid works, and what information can be extracted from it. The case study is a C program with four functions, to show how the Robustness Grid deals with different possibilities of

Language rules application. This chapter will explain how the Robustness Grid shows the robustness measurement results.

Chapter 6 will analyse and evaluate the Robustness Grid in further details. The analysis and part of the evaluation is based on the case study presented in Chapter 5. The evaluation is based on a comparison between the Robustness Grid and previous frameworks that use the same language rules.

Chapter 7 concludes the work that was described in this thesis and reviews the criteria of success that were made in Chapter 1. It will also include some suggestions that could be made for the Robustness Grid in future work.

Chapter Two

Background

2.1 Introduction

In this research, the Robustness of a C program will be measured using language features and a Program Slicing technique. In this literature review, Robustness and Robustness Measurement techniques are defined. Also, the C language features and standards are investigated.

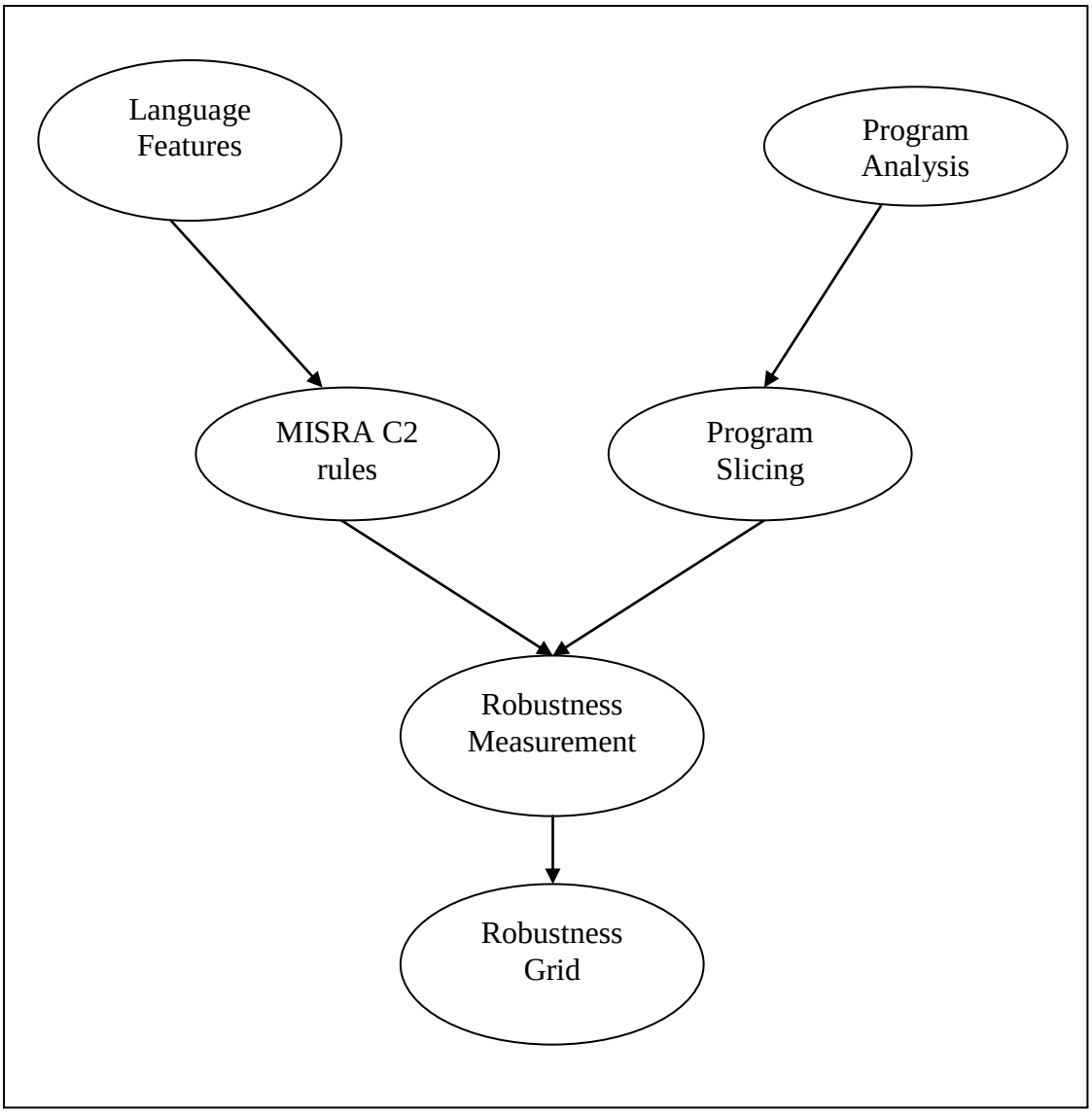

Figure 2.1 Literature Hierarchy

2.2 Program Robustness

Robust: This word is used in many contexts. One instance is that it describes computer software, and at another time it expresses a machine attribute, a mathematical equation, a medicine or a patient. The question is: *What is Robustness?* In this study, the answer to this question comes from a software engineering point of view. Thus, the thesis will concentrate on Software Robustness.

Before defining the meaning of software Robustness, some terms need to be defined to help develop a better understanding of Software Robustness. Correctness,

Dependability, and Reliability will be clarified to differentiate between them and Robustness. They can be understood in many different ways, which makes the definition of Robustness ambiguous, and this thesis gives an unambiguous definition.

2.2.1 Software Dependability

There is a relation between Software Robustness and Software Correctness, Software Dependability, and Software Reliability. Software Correctness may be considered one of the Robustness characteristics. Software Correctness is defined in the IEEE standard (IEEE 1990) as "*The degree to which software, documents, or other items meet user needs and expectations, whether specified or not*."

This definition discussed the software correctness via input and output validity. Here, the only criteria for the evaluation of software correctness are requirements satisfaction, whether they are user requirements or other program specifications.

The opposite side of correctness is the failure situation, where the program has some faults. A fault (IEEE 1990) is: "*a defect in hardware device or component*". In computer programs, a fault means: "*an incorrect step, process, or data definition. 'Bug' and 'error' are common use to express program fault*" (IEEE 1990). Device faults or program faults could cause a program failure (IEEE 1990), which is "*the inability of a system or component to perform its required functions within specified performance requirements*".

There are different classifications for faults. Laprie (Laprie, Arlat et al. 1990) has classified faults depending on the perspective of the: phenomenological cause, nature, phase of creation or occurrence, situation with respect to program boundaries, and persistence.

Software Dependability in general is "*the ability to deliver service that can justifiably be trusted*" (Avizienis, Laprie et al. 2004). This means that the program can avoid failures and it is less likely to be broken or stopped. Furthermore, if Program A

depends on Program B, the dependability of A is affected by the dependability of B (Avizienis, Laprie et al. 2004).

Software Dependability is discussed in the literature and it has been integrated into 6 main attributes (Avizienis, Laprie et al. 2004; Jawadekar 2004; Sommerville 2008; Pressman 2009):

1- Availability.

2- Reliability.

3- Safety.

4- Confidentiality.

5- Integrity.

6- Maintainability.

As shown in Figure 2.2, Security is related to the Dependability attributes. Security and Dependability specifications should include the requirements to produce a robust program.

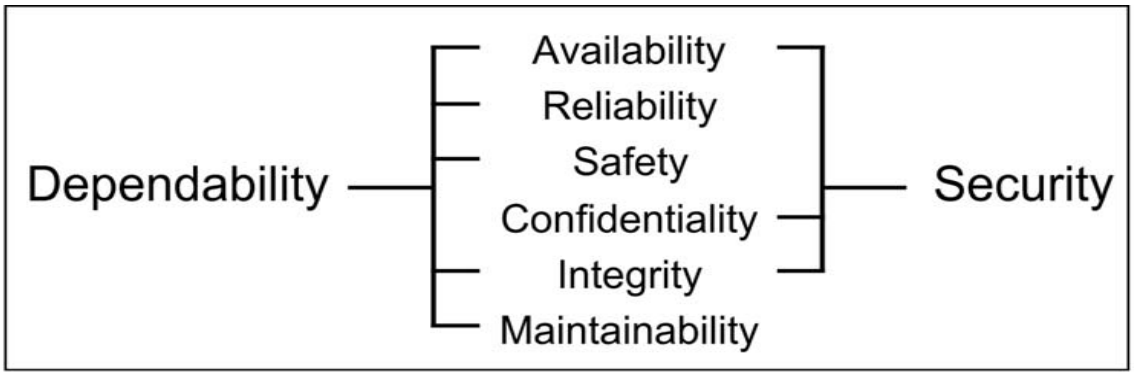

Figure 2.2 Dependability and Security attributes (Avizienis, Laprie et al. 2004)

Avizientis et al. (Avizienis, Laprie et al. 2004) stated Robustness is "*specialised secondary attributes*" for Dependability. It characterises the program reactions towards some faults.

The IEEE definition of Robustness (IEEE 1990) is: "*The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions"*.

In this definition, there are three main aspects; the correct program response, the input data, and the program environment. Program response means that the program should respond rationally (Musa, Iannino et al. 1987), but not necessarily correctly. It should not fail to reply or react illogically. The input data is one of the factors that affect the Robustness of the program. A robust program can continue to operate correctly despite the introduction of invalid input (Pullum 2001).

The environment where the program is run consists of hardware, other software programs, and the humans that interact with the program. These factors also affect program Robustness.

Gribble (Gribble 2001) states that Robustness is "*the ability of a system to continue to operate correctly across the wide range of operational conditions, and fails gracefully outside the range*". Robustness is required in safety critical programs, where program failure could cause massive extreme problems (Weinberg 1983). In the previous definitions, Gribble did not disallow program faults, but the required condition is that the program *fails gracefully* (Gribble 2001), which means that the failure of the program will not cause it to crash or hang. Gribble's definition covers hardware faults (i.e., shortage in power supply). Hardware defects can be considered as *stressful environment conditions.*

2.2.2 Robustness Techniques

Researchers who deal with software Robustness try to develop techniques to build a robust program. On the other hand, others try to find techniques that certify programs and determine whether they are robust.

Robustness can be internal or external (DeVale and Koopman 2002), where internal Robustness is the code of the program (functions, classes, threads, etc), and external Robustness is the surrounding environment (Dabek, Zeldovich et al. 2002).

Arup and Daniel (Arup and Daniel 1997) presented features, such as portability, to evaluate some existing benchmarks of Unix programs. As a result, they built a hierarchy structured benchmark to identify Robustness issues that had not been detected before. Eslamnour and Ali (Eslamnour and Ali 2009) introduced a theoretical foundation for robust matrices that reduce the uncertainty in distributed program.

Different techniques were developed to satisfy Software Robustness. These techniques also utilized different theories and methods to measure Software Robustness.

2.2.2.1 Fault Tolerance

Fault Tolerance and Robustness have the same objective; to make sure program faults do not cause program failure. Both Fault Tolerance and Robustness are needed in all programs, especially in safety critical ones where program failure can cause massive problems.

Fault Tolerance, which provides a program that complies with its specifications in spite of faults, is less costly than other redundancy techniques, but it has the same problem of increased code size and reduced performance (Rebaudengo, Sonza Reorda et al. 1999).

Fault Tolerance can be *Hardware Fault Tolerance* or *Software Fault Tolerance*. *Hardware Fault Tolerance* considers the correctness of the hardware (physical) parts of the program. *Software Fault Tolerance* discusses the correctness of the code. Therefore, one of the advantages of the Fault Tolerance technique is that it can be used to validate any kind of programs (Lyu, Zubin et al. 2003).

Fault tolerant programs can continue in operation after some program faults have occurred. Fault tolerance has four aspects (Sommerville 2008):

1. *Fault detection*: faults that could cause program failure will be detected.
2. *Damage assessment:* the affected program parts will be identified.
3. *Fault recovery:* can be done in two ways. “Backward error recovery” where the program will return to the last constant state (safe state), and “Forward error recovery” where the program repairs the faults and keeps running.
4. *Fault repair:* includes the faults that are not cured in the *fault recovery* aspect.

Implementation of fault tolerance is possible by including checks and recovery action in the software. This is called *defensive programming*. Defensive programming cannot effectively manage program faults, which occur due to the interaction between software and hardware. Software Fault Tolerance has two approaches (Sommerville 2008):

1. N-version programming: using at least three versions of software, this should be consistent in the event of a single failure. In this approach, a different version of the software is run in parallel on different computers. Using a voting program, the program compares the output and the invalid output, which has the least votes or is the latest output, will be rejected.
2. Recovery blocks (RcB): Recovery blocks are dynamic techniques. The program adjusts the output during program execution depending on the Acceptance Test (AT) and backward recovery, where the program returns to the last acceptance stage before the fault happened.

Pullum (Pullum 2001) states that the relationship between software Robustness and Fault Tolerance are mainly those techniques which can handle the following:

- Out of range input.
- Input of the wrong type
- Input in the wrong format.

Pullum also added that robust programs put a mark on the faults, to make it easier for other programs to fix them. Also, software Robustness may have some features shared with fault tolerance techniques, such as testing input type, testing the control sequence, and testing the functions of the process.

2.2.2.2 Redundancy

Redundancy is one of the ideas applied to build component robust software (Huhns and Holderfield 2002). The idea of redundancy is to add a different component, but one that will be equivalent in functionality to old ones. Then, if one part fails to perform correctly, it will be replaced with another that can provide the same services.

Redundancy was used in *hardware* programs such as NASA satellites, by duplicating important hardware subprograms. In *software* programs, however, redundancy cannot be applied in the same way because identical software subprograms fail in the same identical ways. Thus, *redundancy* must be applied to software in a different way (Dix and Hofmann 2002; Huhns and Holderfield 2002).

The challenge in software programs is to design subprograms that can perform and behave in equivalent functionality, but do not fail in the same situations (Huhns and Holderfield 2002; Huhns, Holderfield et al. 2003).

2.2.2.3 Self-adaptive systems

Another method applied to get Robustness software is *self-adaptive software*: where the program has the ability to fix itself. The mechanism is easy to understand, but

difficult to apply. The self-adaptive program can evaluate its work and change behaviour when the evaluation indicates that the program has not done what was supposed to be done. Moreover, a self-adaptive program can fix itself, by doing an alternate behaviour (Laddaga 1999; Mazeiar and Ladan 2009).

Self-adaptive programs do struggle, however, in evaluating functionality and performance at run time, where the evaluation of the outcomes and expectations determination takes time. In addition, self-adaptive programs may manage to get close to the solution of a problem because the program chose the preselected design – time compromise instead of running the optimal or near optimal algorithm for the input and state context at the run time (Laddaga 1999).

Another adaptive program called the "*Self-controlling software model*" was developed. This program contains three loops. The *feedback loop* adjusts program variables to meet the quality of service. The a*daption loop* evaluates the behaviour and performance of the model, and, if necessary, triggers change. The *reconfiguration loop* runs the adaption loop request. Since the reconfiguration loop could include structural changes, it relatively costs more compared with feedback and adaption loops (Mieczyslaw, Kenneth et al. 1999).

2.2.2.4 Event-driven Programming

Event-driven programming is applied in many applications: user interfaces, discrete programs and business module simulations (Dabek, Zeldovich et al. 2002). "*An important characteristic of event-driven computation is that control is relinquished to a library that waits for events to occur. Each event is then dispatched to the application by invoking a handler function or a handler object for appropriate action*" (Petitpierre and Eliëns 2002).

Event-driven programming can be done in three ways (Petitpierre and Eliëns 2002):

- *Event loops*: explicitly dispatching on an event, e.g., completion of a disk transfer, to raise the appropriate applications code.
- *Callback functions*: implicitly dispatching based on an association between a callback function and the type of event. Callback is registered when a program cannot complete an operation because it has to wait for an event. A callback executes indivisibly until it hits a blocking operation, and then, it registers a new callback and returns (Dabek, Zeldovich et al. 2002).
- *Listener objects*: callback on objects with *hook methods* that are invoked on the occurrence of an event. Listener objects are more powerful than callback functions since they must rely on ad-hoc mechanisms to take the history of event occurrences into account.

Event-driven programming is a technique that the user can use to trigger a program in arbitrary order (Philip 1998). The characteristics of event-driven programming encouraged software engineers to use it to obtain a robust program.

Event-based programming can provide a convenient programming model, which may also be extended to take advantage of multi-processors. Debek et al. (Dabek, Zeldovich et al. 2002) concluded that Events are better for managing I/O concurrency in server software than threads because Evens have less complexity and produce more robust software. Also, event-driven programming has an advantage over threads in that event-driven programming provides a convenient programming model which is naturally robust. Debek et al. (Dabek, Zeldovich et al. 2002) added that the event-driven model can be extended to exploit multi-processors with minor changes of the code. However, event-driven program structure has a series of small callback functions, which rely heavily on dynamic memory.

Shahroni and Feldt (Shahrokni and Feldt 2010) introduced a framework, ROAST, to specify the Robustness requirements of a software by categorising requirements into

patterns in different levels. There are three main ideas behind their method; the software specification levels, requirements patterns, and alignment from requirements to testing. Their evaluation showed that the requirements carried out by ROAST are more likely to be verified.

In further work, Shahroni and Feldt (Shahrokni and Feldt 2011) presented a frame work called RobusTest. This framework tests the Robustness properties of a program focusing on timing issues. RobusTest uses the requirements patterns categories, which were introduced in their previous work, to set some test cases to identify the errors in the requirements.

In the above methods, the techniques were used to build a recovery system that can keep the program running and producing an accurate output despite failure. Therefore, the techniques focused on the input/output relations. The program syntax code was almost ignored, and the developers were supposed to use the programming language standards to write their programs. Still, the standards can be applied differently by different developers.

If the program is written following a standard or some rules, the errors and failures will be minimised and the program complexity will be reduced. Therefore, program robustness should start by writing a robust code.

2.2.3 Robustness Testing

Robustness testing checks whether the robust programming techniques have succeeded in satisfying the Robustness conditions certified for the program. The main term used in robustness testing is that the program should continue the normal function despite the invalid input, or it should fail gracefully.

Testing can only reveal Robustness errors in *successful test cases.* In addition to robustness testing, there are other important definitions: A *robust error* is defined as an inrobust reaction to a test case produced during its execution. *Inrobust reactions*

are observed when a test objects crash or hang. A *test object* in this context is a software component tested for its robustness (IEEE 1990; Dix and Hofmann 2002).

The importance of software Robustness drives researchers to develop different techniques and tools to test software Robustness. Some testing software Robustness tools and techniques are discussed below.

2.2.3.1 Interface Robustness testing tools

Interface Robustness testing is where the success criteria in most of the cases is "*if it does not crash or hang, then it's robust*" (Koopman, Devale et al. 2008).

- Fuzz (Miller, Koski et al. 1995): is an automatic and simple method where a random input stream is used as a Robustness testing method. Nine versions of the UNIX operating program and X-Window applications were tested using this method. The failures were identified and categorized: *crash* (with core dump) or *hang* (infinite loop).

  The results show that over 40% (in the worst case) of the basic programs and over 25% of the X-Window applications crashed or hung. They were not able to crash any of the network services that they tested or any of the X-Window servers.

- The Riddle tool (Schmid and Hill 1999): is a tool used to test the Robustness of Windows NT. Two different approaches were examined in this paper to generate data (generic data generation, and intelligent data generation) to be used for automated Robustness testing. They concluded that this tool is useful for constructing both generic data and intelligent data, where they discovered new kinds of failures.

- Ballista (DeVale, Koopman et al. 1999; Koopman 2002): is an automated Robustness testing tool designed to exercise commercial off-the-shelf (COTS)

software components. Ballista is a methodology and web server that remotely tests software modules in linkable object code form. Ballista's purpose is to identify sets of input parameters that cause Robustness failure in the software components being tested. Ballista testing begins with identifying the data types used by an API (Application Programming Interface) under test. Application-specific data types can inherit base test cases from predefined data types in the Ballista testing tool set. Then, the Ballista test harness generator is given the signature of a function to be tested in terms of those data types, and it generates a customised testing harness. The test harness composes combinations of test values for each parameter and reports Robustness testing results.

2.2.3.2 Dependability benchmark Robustness testing tools:

A Dependability Benchmark defines benchmarks to characterise the program behaviour under normal loads and faults. The goal of benchmarking the dependability of computer programs is to provide generic ways for characterising their behaviour in the presence of faults (Kanoun, Madeira et al. 2002). There are some tools that used this technology to develop a Robustness tester tool:

- DBench: The DBench project aims at defining a conceptual framework and an experimental environment for dependability benchmarking (Kanoun, Madeira et al. 2002).
- Autonomic Computing benchmark: evaluate a computing system along the four core autonomic dimensions of self-healing, self-configuration, self-optimization, and self-protection (Brown, Hellerstein et al. 2004; Brown and Redlin 2005).

In the DBench and Autonomic Computing benchmark robustness testing techniques, the program is tested using different or random test cases. The test cases depend on the program execution and whether it succeeds or fails to deliver a robust output or fails nicely. The program has to be tested before execution to reduce the errors that

can be made by imprecise programming syntax format or defective program data flow.

2.2.4 Robustness Measurement

Measurement as an activity is used in everyday life; in the supermarkets, clothing stores, and driving journeys, where the prices, sizes, distances and other aspects are measured to help in decision making. In Software measurement, many things can be measured, but the question is: how to measure Software?

In the UK, the term Software Measurement is also known as Software Metrics, Software Engineering measurement, or Software Metrication (Zuse 1998).

In general, Measurement means "*the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules*" (Fenton and Pfleeger 1997). The entities are the object to be measured, and the attributes are the object features or properties.

By applying this definition to Software, it will lead to Software Measurement, which is "*a quantified attribute of a characteristic of a software product or the software process*" (ISO/IEC 2007). Software measurement could mean estimating the cost, determining the quality, or predicting the maintainability (Fenton and Pfleeger 1997).

Robustness can be also measured. By using the Measurement definition, the Robustness Measurement can be defined as the process by which relative numbers are assigned to Robustness Degree of a program in such a way to describe them according to standard rules.

There are different Measurement classifications, depending on the features or the programs that are measured. There are measurement techniques for Software Quality, Software Complexity, Software Validity, and for Object Oriented programs,

which can be also classified into Static and Dynamic Measurements (Kaur, Minhas et al. 2009). In this research, only Robustness Measurement is discussed further.

Safety critical programs must be robust to avoid the problems that could be caused by failures (Jones 2009). Several techniques have been tried to measure program Robustness.

In order to obtain a measure of programs, they have to be analysed. The analysis can be either static or dynamic. Static analysis looks at the programs without using any input or executing the code (Ayewah, Hovemeyer et al. 2008). On the other hand, dynamic analysis looks at the program execution behaviour and input/output relation. Measuring the application of a language standard to a program is a static analysis technique.

Extended Propagation Analysis (EPA) (Voas, Charron et al. 1997) is an example of a robustness measurement using the Fault Injection technique. EPA predicts how software will behave when a component fails due to the effect of an external component failure, an invalid input or an unlikely operational environment.

EPA is only concerned with the software behaviour and output. In one of the case studies mentioned in (Voas, Charron et al. 1997), a module called Yaw, which is part of the 737 aircraft controller system, was measured. In Yaw, the input values were taken from NASA. The Yaw module was run twice; the first time with normal data and the second time with data with faults injected into it. The fault tolerance measurement using EPA showed that the total failure tolerance of the Yaw program estimated that 43.5% of the input data can cause a failure.

Voes et al. concluded that the EPA technique does not test the correct behaviour of a program. However, it identifies the code locations that reduce the robustness. It also measures an acceptable level of robustness in a program. EPA does not guarantee a

robust behaviour of a program in the future, but it gives an indication for such behaviour.

Hamann et al. (Hamann, Racu et al. 2007) used some Robustness criteria, such as input data rate and CPU clock rate, to create multi-dimensional Robustness matrices and use them to measure the Robustness of a program.

Hamman et al. introduced two-dimensional robustness metrics; static and dynamic. The static robustness metric is done in the design phase of the program and cannot be changed later. The dynamic dimension is made in the execution phase and can be used to modify the program to increase its robustness.

The above robustness measurement techniques do not give the developer a fully detailed measurement. Also, they do not specify the part of programs that need to be modified to raise the quality of the program.

2.2.5 Summary

Program Robustness can be measured from different perspective and viewpoints. In the literature reviewed in this section, different Robustness definitions were explored to find one definition to be used further in this research. Robustness measurement is a scale that shows how the program is robust according to some standard. In this research; however, a new method will be introduced to measure the Robustness that was defined earlier.

2.3 C Language Standard

The C Language standards were introduced to avoid the code misinterpretation, misuse, or misunderstanding. The IEEE has the ISO/IEC 9899:1999 standard (ISO/IEC 1999) for the C language, which was used later by MISRA to produce MISRA C1 and C2. This in turn led to Jones producing “The New C Standard: An

Economic and Cultural Commentary" (Jones 2009). Other C standards such as "C programming language Coding guideline" (Laroche 1998) are less frequently used.

2.3.1 ISO standard

International Organization for Standardisation (ISO) has published international standard for Business, Government, and Society (ISO/IEC 1999). Some of these standards are for Software Engineering and for Programming languages.

In this research, the ISO/IEC 9899:1999 C standard (ISO/IEC 1999) has been used to develop a new Robustness Measurement technique. This standard was published in 1999 followed by three "Technical Corrigenda" in 2001, 2004, and 2007. This international standard was designed to promote the portability of C programs. It is intended for use by implementers of compilers and programmers (American National Standards Institute (ANSI) 1999).

The ISO/IEC 9899 standards specify the representation, syntax, and constraints of the C language. Also, they specify the semantic rules for interpreting C programs. In the standard syntax, a set of rules was introduced to show the recommended way of writing the C language notations, and the methods of identifying the language concepts such as identifiers scope, linkage, name space, and types. The standard also shows the allowed and prohibited type conversion in the C language. The C lexical elements are listed, such as keywords, and other C language syntax constructions, such as external definitions, and the proper way to write and use them is clarified with examples.

The semantic part of the standard stated the proper use and interpretation of each construction in the C language syntax.

ISO/IEC gives recommendations for the representation of processed input data and produced output data of the C program. In addition, it describes the limitations of the C program implementations. On the other side, the standards do not specify how the

input/output data is going to be used in the program, or indicate the complexity level of the program code (ISO 2012).

2.3.2 MISRA C Language Rules

The Motor Industry Software Reliability Association (MISRA) has published a standard set of rules for C and C++ "*to provide assistance to the automotive industry in the application and creation within vehicle programs of safe and reliable software*" (MISRA 2012). MISRA C 1998 rules ("MISRA C1") where published in 1998 and were followed by a technical clarification document in 2000. In 2004, MISRA published a second version of MISRA C rules (MISRA C2) to address some technical and logical problems, and for further technical clarification. In MISRA C2, the rules are rephrased to be more sensible, accurate and comprehensive.

MISRA C2 rules are classified into two types: Required (122 rules) and Advisory (20 rules). Required rules are obligatory and must be followed by developers to create safe programs, and in general, the violation of these required rules leads to a system failure. Advisory rules are necessary but not as important as the Required rules. However a developer should follow the advisories in order to build a safe program.

In MISRA C2, the rules are categorised in 21 categories. The MISRA categories cover all the C language common programming issues such as programming processes, coding styles, and programming syntax. The MISRA categories start with the *Environment* category, which describes the optimum environment for C programs. Then, there is the *Language extensions* category, which has guidelines on how the comments should be written in the program code. The *Documentation* category contains general rules for the documentation process.

The rest of the MISRA C2 categories cover the language syntax format. The categories cover data structures, such as arrays and pointers, format and use. Also, flow control and type definition and conversion rules are listed, in addition to the

function and initializing format. Each of these categories has a set of rules that give instructions on the way the code syntax should be written.

An example of a MISRA C2 rule is Rule 8.1:

> *Rule 8.1 (required) Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call*. (MISRA 2004)

"X.y" is the MISRA rule numbering method and means this is Rule 1 ("y") in Category 8 ("X") (Declarations and definitions). "*required"* means the rule is an obligatory rule.

There are 3 tools that use MISRA C2 rules to test the program robustness; these tools are explored to find out how they work. The tools are: TBmisra, FlexeLint, and Klocwork Truepath.

The Liverpool Data Research Associates (LDRA) Company has several tools to evaluate the robustness of C programs. These tools evaluate the C program against MISRA C2, ISO 1990, and LDRA rules. LDRA has produced LDRA TBmisra, which is test C program against the MISRA C and C2 rules.

The LDRA TBmisra evaluation classifies errors into three types depending on their importance level. These rules are displayed in the evaluation results as symbols (M, C, and O) are:

1- Mandatory Rules (M): if a rule of the mandatory type is violated, the evaluation will report a fault.

2- Checking Rules (C): the violation of these rules may cause different implementation of the program using different compilers or platforms. The Checking rules will give the developer a chance to make sure that the program has consistency over all different platforms.

3- Optional Rules (O): the developer has the option of fixing the violation of these rules.

In addition to the above error types, the LDRA analyses C program using Static and Dynamic analysis. The Static analysis in LDRA is classified depending on code type. These types are (LDRA 2012):

1- Programming Standards Verification: Assesses whether the source code conforms to a set of user-configurable programming standards.

2- Structured Programming Verification: Reports on whether the source code is properly structured.

3- Complexity Metric Production: Reports on a number of complexity metrics.

4- Full Variable Cross Reference: Examines and reports global and local variable usage within and across procedures and file boundaries.

5- Unreachable Code Reporting: Reports on areas of redundant code.

6- Static Data Flow Analysis: Follows variables through the source code and reports any anomalous use.

7- Information Flow Analysis: Analyses inter-dependencies of variables for all paths through the code.

8- Loop Analysis: Reports the looping structure and depth of nesting within the code.

9- Analysis of Recursive Procedures: All the static analyses in LDRA are performed individually and on sets of mutually recursive procedures.

10- Procedure Interface Analysis: The interface for each procedure is analysed for defects and deficiencies. The interfaces are then projected through the call graph of a system to highlight integration defects.

The Dynamic analysis covers other areas such as (LDRA 2012):

1- Statement Coverage: covers all the statement that related to the selected statement.

2- Branch/Decision Coverage: covers all the control that related to the selected statement.

In addition to TBmisra, LDRA has a tool called **TBvision**. The TBvision tool is a graphical presentation of an error report. The graphical screen shows a summary of the rules that were violated in the program. In TBvision, the developer can access the code and the TBmisra report that uses it.

FlexeLint, developed by Gimple Software, is a tool that does a static analysis for C/C++ programs, and uses their own rules in addition to the MISRA C2 rules (Gimple Software 2012). FlexeLint do not use a compiler, so in the test result there are some rules violated that can cause a run time error in the compiler.

In addition to the FlexeLint violated types and numbers, these types are:

1- Syntax Error (Error).

2- Internal Error (Intern).

3- Fatal Error (Fatal).

4- Warning (Warning).

5- Informational (Info).

6- Elective Note (Note).

The Klocwork Truepath (Klocwork 2012) is a tool that uses static analysis to assess C programs. It uses the MISRA C2 rules in addition to other rules such as C CERT. There are 21 of the MISRA C2 rules that are not supported by Klocwork Truepath, 15 of them are not verified, and the rest are not supported even though they are verified.

2.3.3 Other C Language Rules

Straker (Straker 1992) introduced general guidelines to develop a robust C program. He left the choice for the programmers to create their own standards using these guidelines. However, he introduced his own interpretation of the guidelines. In his standards, Maguire (Maguire 1993) covered main issues, such as dealing with files, commenting in the program, and naming. The guidelines give the developer advice of how to write bug-free code, and how to avoid the common mistakes that cause program faults. For example, in this advice: "*As you step through code, focus on data flow*" (Maguire 1993), he gave the developer an advice about the data flow. This advice is a conclusion of an example he showed that illustrated the importance of data flow. The main aim was to help the developer write bug-free programs before the testing phase. However, the Maguire guidelines were written in an abstract way and each programming issue was discussed with examples to make sure that developers will understand how and when to use these standards.

The Software Engineering Institute has introduced C, C++, and Java language standards. CERT C is the C programming Language standard, and "*rules and recommendations for secure coding in the C programming language"* (Seacord 2012). The goal of these rules and recommendations is similar to other rules and standards, which is "*to eliminate insecure coding practices and undefined behaviours that can lead to exploitable vulnerabilities. The application of the secure coding standard will lead to higher-quality systems that are robust and more resistant to attack*" (Seacord 2012).

In CERT C, the program rules are given a value of priority and level. The priorities are "*assigned using a metric based on Failure Mode, Effects, and Criticality Analysis*". The priorities have a scale for Severity, Likelihood, and Remediation Cost. Each scale is from 1 (low) to 3 (high). The three scales together will produce a

multiplicative scale from 1 to 27, and the levels depend on this scale where Level 1; the highest, contains the values of (12, 18, 27) for the multiplicative scale of Severity, Likelihood, and Remediation Cost, where Level 2 has (6, 8, 9), and Level 3, the lowest, has values of (1, 2, 3, 4). For example, if a rule scores: 2 in Severity, 3 in Likelihood, and 1 in Remediation cost, then the priority will be 2X3X1 = 6, which means it is in Level 2. (Thompson 2010).

In C CERT, the rule is described first. Then, a non-compliant example about this rule is given, followed by a compliant example for the same rule. After that, the exception, if there is any, is shown and the risk assessment regarding to the calculations mentioned before are given. Below is an example that shows one of the rules:

"*Do not use the same variable name in two scopes where one scope is contained in another*" (Software Engineering Institute 2011).

Laroche (Laroche 1998) aims in the "*C programming language Coding guideline*" to make the code less defective, more robust (against changes in code architecture), and more readable (for easier maintenance). "*The variable scope should be as small as possible*" is an example of the guidelines.

2.3.4 Summary

C standards in general aim to produce robust programs with fewer errors. Also, the standards try to make the code readable and clear to all different developers to avoid code misinterpretation or misuse.

The MISRA rules were chosen in this research because the MISRA C2 rules implicitly include the ISO and ANSI standard. They are also simple to be apply and understand since they are only 142 rules written in English by people who use C language in their work. Since there are different companies and tools that use and apply the MISRA C2 rules, the application of the rules can be evaluated.

Other rules such as C CERT are not as simple to apply since they contain many rules and this is the case for ISO as well. In the guidelines, the rules are very abstract and unsuitable to measure the Robustness.

2.4 Program Slicing

Weiser (Weiser 1979) introduced Program Slicing as "*a method used for abstracting from computer programs*". The slice of program **P** with respect of the slicing criteria Slice **S** <***L, V***>, where ***L*** is a statement line number in **P** and **V** a variable (Binkley and Gallagher 1996).

Different types of program slicing have been developed and these can be classified into various types depending on different criteria. Some program slicing techniques will be addressed in details below.

2.4.1 Static program slicing

Program slicing was introduced first as a Static Slice. Static means that only statically available information is used for computing slices (i.e., all possible executions of the program are taken into account) (Baowen, Ju et al. 2005). A Static slice is constructed by assigning a *point of interest* and deleting all irrelevant statements to this point (Weiser 1981).

A point of interest is the variable in a specific place in the program that is going to be sliced. It is signed by the variable **V** and the line number **L**. This called the *slicing criteria* and is expressed as **S <L, V>**, where **S** is the slice we are interested in (Tip 1995; Harman and Hierons 2001). Static slicing is considered as code preserving analysis, where it only retrieves the code lines without any change on their syntax (Gallagher and Binkley 2008).

A Static slice can be executable or non-executable (Tip 1995). An executable slice means the code that was produced after the slicing operation (the slice) can be compiled and run as a program.

Static slicing has many types. In this literature review, the most frequently static slicing types used are: Backward, Forward, Conditioned, Decomposition, Amorphous, and Quasi Slicing.

```
1 int x = a;
2 int y = 25;
3 int z = 0;
4 for (int i = 0; i < x; i++){
5     z = z + y;
6     y = y + 2*i;       }
7 printf ("Y is: %d", y);
```

Figure 2.3 Original program P1 (Kim and Fong 2007)

2.4.1.1 Backward Slicing

Weiser (Weiser 1979) introduced the program slicing which was later known as Executable Backward Static Slicing. It is Executable because the slice produced is an executable program (i.e., without considering the program input) (Binkley and Gallagher 1996).

Backward Slicing uses the same slicing criteria as the Static Slicing, where Slice **S** is retrieved using slicing criteria **<L, V>**, where **L** is the statement line number and **V** is the variable name. A Backward slice is computed by gathering statements and control predicts by a backward traversal of the program starting at the slicing criteria (Tip 1995). Backward slicing contains the statements of the program which have effect on the criteria slice and answer the question "*what program components might effect a selected computation?"* (Gallagher and Binkley 2008)

As an example, Figure 2.3 shows a program **P1** that is going to be sliced.

```
1  int x = a;
2  int y = 25;
4  for (int i = 0; i < x; i++){
6     y = y + 2*i;            }
7  printf (“Y is: %d”, y);
```

Figure 2.4 Backward slicing on (y,7) in P1 (Kim and Fong 2007)

As shown in Figure 2.4, applying backward slicing on (7, y) in program ***P1***, will delete all statements that have no effect on Statement 7. So, Statements 3 and 5 are deleted.

A static backward slice preserves the meaning of the variable(s) in the slicing criterion for all possible inputs to the program (Gallagher and Binkley 2008).

2.4.1.2 Forward Slicing

Forward Slicing uses the same slicing criteria **<L, V>** as the static slicing technique. However, the Forward Slice answers the question “*what program components might be effected by a selected computation?”* (Gallagher and Binkley 2008)

A Forward slice captures the impact of its slicing criteria and it is considered a kind of wave effect analysis (Black 2001; Baowen, Ju et al. 2005). It contains the set of statements and control that were affected by the computation of the slicing criterion that was computed by the variable **V** at the program point or line number **L** (Horwitz, Reps et al. 1990; Tip 1995; DeLucia 2001; Harman and Hierons 2001).

```
1 int x = a;
2 int y = 25;
3 int z = 0;
4 for (int i = 0; i < x; i++){
5     z = z + y;
6     y = y + 2*i;          }
7 printf (“Y is: %d”, y);
```

Figure 2.5 Forward slicing on (y,2) in P1 (Kim and Fong 2007)

Forward Slicing of Program **P1** in Figure 2.3 is shown in Figure 2.5. It produces the same statements of the **P1** program because Statement s2 affects all program statements. Therefore, the produced slice will contain all statements. The challenge that faces Forward slicing is to produce an executable slice, where it is difficult for forward slicing to preserve the semantic of a executable code (Binkley, Danicic et al. 2006; Kim and Fong 2007).

In Forward Slicing a statement is computed depending on the values computed in the Slicing Criteria (Tip 1995). Forward and Backward slicing is computed in the same way, where they use the same slicing criteria. However, the direction of code analysis is the difference between them.

In addition, Binkley and Harman (Binkley and Harman 2005) proved that "*For a large class of programs, the distribution of forward slices will contain a significantly larger proportion of small slices when compared to the distribution of backward slices*."

2.4.1.3 Conditioned slicing

Conditioned slicing "*consists of a subset of program statements which preserves the behaviour of the original program with respect to a slicing criterion for a given set of execution paths*"(Canfora, Cimitile et al. 1998). The Slicing Criteria of the conditioned slice is **<L, V, C>**, where **L** is the line number, **V** is the variable name, and **C** is the condition. The conditioned slice isolates the code that semantically satisfies the slicing criteria condition (Baowen, Ju et al. 2005; Gallagher and Binkley 2008).

The condition in the slicing criteria, which could be an input value of a variable, allows the user to fragment a program from different angles or using different input data (Canfora, Cimitile et al. 1998; DeLucia 2001).

Danicic et al. (Danicic, Fox et al. 2000) implemented a conditioned slicer (ConSIT) based on conventional static slicing, symbolic execution and theorem proving.

2.4.1.4 Decomposition Slicing

Decomposition slicing is a slice used to decompose a program into different components. Decomposition slicing is a union of certain slices taken at certain line numbers on a given variable (Gallagher and Lyle 1991). Decomposition slicing does not use a line number in the Slicing criteria, so the Slicing criterion of it is only the variable name **<V>**.

```
1 #define Yes 1
2 #define No 0
3 main()
4 {
5     int c, nl, nw, nc, inword;
6     inword = NO;
7     nl = 0;
8     nw = 0;
9     nc = 0;
10    c = getchar ();
11    while(c != EOF) {
12      nc = nc + 1;
13      if (c == ‘\n’)
14        nl = nl +1;
15    if (c == ‘ ’|| c == ‘\n’|| c == ‘\t’)
16          inword = NO;
17    else if (inword == NO) {
18          inword = YES;
19          nw = nw + 1;
20    }
21          c = getchar ();
22    }
23          printf(“%d \n”, nl);
24          printf(“%d \n”, nw);
25          printf(“%d \n”, nc);
26 }
```

Figure 2.6 Original Program P2 (Gallagher and Lyle 1991)

Decomposition slicing has two parts: *The slice* and the *complement*. *The slice* “*captures all relevant computations involving a given variable*” (Gallagher and Lyle 1991), where a decomposition slice depends only on the variable name, and does not depend on statement number. *The complement* is the rest of the program code that is not included in the slice (Gallagher and Lyle 1991).

Figure 2.7 shows a decomposition slice of Program **P2** (shown in Figure 2.6) with the slicing criteria (nc). In this slice, all statements that affect variable **nc** are included in the slice, and also all statements that are affected by variable **nc**.

The complement (Figure 2.8) contains all statements that affect other variables, and not related to variable **nc**.

```
3     main()
4     {
5           int c, nc;
9           nc = 0
10          c = getchar ();
11          while (c != EOF) {
12                nc = nc + 1;
21                c = getchar ();
22          }
25          printf (“%d \n”, nc);
26    }
```

Figure 2.7 Decomposition slicing on (nc,26) in P2 (Gallagher and Lyle 1991)

Decomposition slicing can categorise program variables into three categories (Gallagher and Lyle 1991): *independent*, *strongly dependant*, and *maximal*. A variable is called an *Independent variable* if its Decomposition slice does not intersect with any other variable’s decomposition slice. In other words, they would share neither control flow nor data flow.

Strongly dependant variable is the variable that its decomposition slice is a part of another variable slice. *The maximal* is if the variable decomposition slice shares some statements with another variable decomposition slice.

```
 1 #define Yes 1
 2 #define No 0
 3 main()
 4 {
 5 int c, nl, nw, nc, inword;
 6    inword = NO;
 7    nl = 0;
 8    nw = 0;

 11   while(c != EOF) {

 13         if (c == '\n')
 14               nl = nl +1;
 15         if (c == ' '|| c == '\n'|| c == '\t')
 16               inword = NO;
 17         else if (inword == NO) {
 18               inword = YES;
 19               nw = nw + 1;
 20         }
 21         c = getchar ();
 22         }
23          printf("%d \n", nl);
 24         printf("%d \n", nw);
 26   }
```

Figure 2.8 The complement of decomposition slicing on (nc,26) in P2 (Gallagher and Lyle 1991)

2.4.1.5 Amorphous Slicing

Amorphous Slicing uses the same Slicing criteria as in static slicing **<L, V>**. However, while the other types of program slicing are *syntax-preserving*, Amorphous Slicing alters the syntax of the slice with respect to preserved semantics (Harman and Danicic 1997). Amorphous slicing may perform any syntax transformation to simplify the slice for preserving program behaviour (Fatiregun, Harman et al. 2005; Gallagher and Binkley 2008). Amorphous slicing has two types: Amorphous static slicing where

it uses the slicing criteria **<L, V>**, and Amorphous conditioned slicing where it uses the conditioned slicing criteria **<L, V, C>** (Harman, Binkley et al. 2003).

2.4.1.6 Quasi Static Slicing

Quasi static slicing was introduced to mix the slicing methods that range between static and dynamic slicing (Venkatesh 1991). Quasi slicing is used in applications where the values of some input variables are fixed while the behaviour of the original program must be analysed when other input values vary (DeLucia 2001). Therefore, the Slicing Criteria is **<L, V, P>**, where the **L** is the line number of variable **V**, and **P** is the list of inputs that can be fixed or vary (Chung, Lee et al. 2001). When all variables are unconstrained, quasi slicing becomes the same as static slicing. When all variables are fixed, quasi slicing is considered the same as dynamic slicing (Baowen, Ju et al. 2005).

Static Slicing techniques mainly use a variable on a certain place in the program as point of interest, and analyse the program to determine which other code lines are affected by the variable. The variable can have some conditions, or assigned to group of input data.

2.4.2 Dynamic Slicing

A Dynamic slice contains all the statements that "*affect the value of a variable at a program point for a particular execution of the program"* (Agrawal and Horgan 1990). In dynamic slicing, a point of interest is the statement to be sliced. It is marked by the line number **L**, the variable **V**, and the input **P <L, V, P>**. The input, **P,** is assigned to some values which produce a slice regarding these input values, where in the static slice, the program at the selected variable is sliced under all these variable inputs (Korel and Laski 1990).

Compared with static slicing, dynamic slicing can significantly reduce the size of the slice, because the run-time information is collected during program execution and used to compute program slices (Korel and Rilling 1998). The example shown in Figure 2.10 demonstrates how a dynamic slice produces a slice which is smaller than the slice produced by static slicing.

```
1 scanf (“%d”, &n);
2 int s = 0;
3 int p = 0;
4 for (int i = 1; i <= n; i++){
5 s += i;
6 p *= i;  }
```

Figure 2.9 Original Program P3 (Kim and Fong 2007)

The **P3** program (Figure 2.9) has a bug, where p in Line 3 should not be zero because it is used later in Line 6 in a multiplication computation.

Two types of slicing were applied on **P3**; Static and Dynamic. Both slicing techniques generated slices different in size, where dynamic slicing is significantly smaller. Static slicing was applied on (p, 6), and it reduced the number of codes lines, but it failed to find the bug. When Dynamic slicing, was applied on (p, 6) and *p=0* as the assigned value, it returned only the statement that contains the bug.

Static slicing	Dynamic slicing

`1 scanf (“%d”, &n);`
`3 int p = 0;`
`4 for (int i = 1; i <= n; i++){`
`6  p*= i;`
	`3 int p = 0;`

Figure 2.10 Difference between Dynamic slice and Static slice on P3 (Kim and Fong 2007)

2.4.2.1 Simultaneous Dynamic Slicing

Simultaneous dynamic slicing is an extension of dynamic program slicing and was introduced by Hall (Hall 1995). Simultaneous dynamic slicing is applied to more than one test case simultaneously which lead to the Slicing Criteria being **<L, V, {P1, P2…Pm}>** where **Pm** is a list of input values (Sasirekh, EdwinRober et al. 2011). The final slice is constructed using dynamic slicing in regard to each behaviour of the program execution input values set (Baowen, Ju et al. 2005). Simultaneous dynamic slicing is used to locate functionality in code, where the set of test cases can be employed to give a specification of the functionality to be identified (DeLucia 2001).

Dynamic Slicing focuses on the execution of the point of interest in a program, and returns the statements that were affected by that particular execution. The dynamic slicing criteria include a variable and its line number with a value assigned to it.

2.4.3 Other Slicing techniques

There are other slicing techniques which are produced by extending other types of slicing or mixing them. Relevant slicing is a technique considered as an extension of dynamic slicing. All statements that make the program executable will take parts of the slice even if they have no effect on the output. This slice is used in incremental regression testing (Agrawal, Horgan et al. 1993).

Chopping and dicing are two themes which are strongly related to program slicing. A program dice (Weiser and Lyle 1986) only shows code that contributed to bad behaviour and did not contribute to good behaviour. Program dicing is used in program debugging to reduce the time of debugging examination. Chopping (Jackson and Rollins 1994; Krinke 2004) solves the problem of how variables affect each other. Chopping shows only code that contributes to bad behaviour and was affected by some given piece of code.

Object oriented and aspect oriented slicing (Larsen and Harrold 1996; Zhao 2002), distributed programs slicing (Gramoli, Vigfusson et al. 1999), web-based application slicing (Junhua, Baowen et al. 2004; Tonella and Ricca 2005), and slicing under UML scenario models (Qian and Xu 2008) are slicing techniques that have been tried and used to analyse the program code in different ways. However, all of these slicing techniques focused on the program variables and their values and none of them has considered the rest of program code syntax.

2.4.4 Program slicing applications

2.4.4.1 Debugging

Program slicing was introduced the first time by Mark Weiserto make program debugging easier (Weiser 1982; Weiser 1984). Slicing helps the developer by reducing the search space if the output of a program is wrong (Weiser and Lyle 1986; Shinji, Akira et al. 2002). This use of slicing in debugging was the motivation to introduce Dynamic slicing (Harman and Hierons 2001).

There are some models and tools that are based on program slicing to aid in program debugging. SPYDER (Agrawal, Demillo et al. 1993) was developed as a debugger based on dynamic slicing and execution backtracking techniques.

2.4.4.2 Regression Testing

Regression testing is "*selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements*" (IEEE 1990).

Program slicing is used in testing to simplify the testing process (Harman and Danicic 1995; Binkley 1998). Regression testing is a testing process used during the maintenance phase, to make sure that the changes only happened to the target code and did not cause any problems in any other program (Binkley 1998).

Dynamic slicing (Binkley 1998; Lalchandani and Mall 2008), and decomposition slicing (Gallagher, Hall et al. 2007), are both used in regression testing to determine whether two components have the same behaviour. Decomposition slicing is also applied to find an approximate solution to idetify a set of affected components by maintenance (Ngah and Gallagher 2009), and to specity a set of tests to examine these components (Binkley 1998; Lalchandani and Mall 2008).

2.4.4.3 Software Maintenance

Most programs spend 70% or more of their life time in the maintenance phase (Gallagher and Binkley 2008). Program slicing can be used to reduce the time and effort spent on program maintenance. Gallagher and Lyle (Gallagher and Lyle 1991) used Decomposition slicing to define the dependability of a variable in a program. If there is any change to the variable, decomposition slicing will define which statements will be affected by this change, by defending the dependability statements of this variable.

Gallagher et al. (Gallagher and Lyle 1998) developed a tool for software maintenance based on decomposition slicing. The *Surgeon's Assistant* helps maintainers and developers of ANSI C programs isolate program components for change or adaptation. Also, it helps in finding the changes in program design and code, and helps in regression testing where it makes sure that the changes do not affect other components.

Another maintenance phase is called *Program comprehension* and is the process in which a programmer understands the program (Gallagher and Binkley 2008). Conditioned slicing (DeLucia, Fasolino et al. 1996) and decomposition slicing (Gallagher and O'Brien 2001; Tonella 2003) can be used to help the programmer identify a set of statements that preserve the program behaviour with respect to a set of program executions.

2.4.4.4 Other program slicing applications

Program slicing is employed in other applications. *Clustering equivalent computation*, program slicing was used to find the dependant cluster and dependant pollution (Binkley and Harman 2005; Gallagher and Binkley 2008).

In model reduction (Hatcliff, Dwyer et al. 2000), program slicing was applied to remove irrelevant code and reduce the size of the corresponding model. While in database schemas (Maule, Emmerich et al. 2008), program slicing can be used to reduce the size of the program that needs to be analysed to identify the impact of relational database schema changes upon object-oriented applications.

In software robustness (Gallagher and Fulton 1999), Decomposition slices are used to determine a unique fault injection point for any given variable of interest at a point where the variable has the highest impact on program output.

2.4.5 Program slicing tools

Researchers apply program slicing by introducing new tools or modifying existing tools.

2.4.5.1 CodeSurfer (CSurf)

GemmaTech company introduced the Code Surfer (CSurf) as a "*automated source-code analysis tool*" (GrammaTech 2012) used to call the program graphs and help in finding the bugs in C language programs.

Code Surfer calculates the representation of the program constructs such as preprocessor directives, and enables them to be explored through graphical user interface or accessed through optional Application Programming Interface (API).

Code Surfer can do different kinds of code analysis, such as impact analysis, dataflow analysis, pointer analysis, and whole program analysis where it shows the interaction between the program files.

2.4.5.2 Indus/Kaveri

Indus is a framework for analysing and slicing concurrent Java programs. Indus presents a collection of advanced features useful for effective slicing of Java programs including: calling-context sensitive slicing, scoped slicing, control slicing, and chopping. Kaveri is an eclipse plug-in front-end for the Indus Java slicer. It utilizes the Indus program slicer to calculate slices of Java programs and then displays the results visually in the editor. The purpose of this project is to create an effective tool for simplifying program understanding, program analysis, program debugging and testing (Ranganath and Hatcliff 2007).

2.4.5.3 JSlice

Another Java Slicing tool is JSlice, which is a "*dynamic slicing tool for Java programs. It collects and analyzes an execution trace (for slicing) in a compressed form*" (Wang and Roychoudhury 2004).

2.4.6 Summary

Program Slicing is a code analysis technique. There are two different Program Slicing techniques that were investigated in this literature reviewed in this section. Static Program Slicing and Dynamic Program Slicing, which have different applications, are summarised in Table 2.1. Different tools were used to slice programs in both languages; C and JAVA.

Slicing Type	Slicing sub-type	Slicing Criteria	Applications
Static Slice	Forward	<L, V>	Program Debugging Database Schemas Analysis
	Backward	<L, V>	Program Debugging Database Schemas Analysis
	Conditioned	<L, V, C>	Program Comprehension Program Maintenance
	Decomposition	<V>	Program Maintenance Program Robustness Program Comprehension Regression Testing
	Amorphous	<L, V>	Program Comprehension
	Quasi Static	<L, V, P>	Program Comprehension
Dynamic Slice		<L, V, P>	Program Debugging Program Comprehension
	Simultaneous	<L, V, P1…Pm>	

Table 2.1 Program Slicing techniques and their applications

2.5 Summary

In this literature review, the Robustness of a program was defined as “the degree to which a system or component can function correctly in the presence of invalid input or stressful environment”. This means that Robustness can be a relative value and can be measured. There are many techniques that have been used to satisfy or define Robustness. These techniques introduce some solutions to create a robust program, such as avoiding the faults by redundancy. However, they are still facing some challenges like the complexity of redundancy.

The MISRA C2 language rules cover most of the C language issues, and by following the MISRA rules most of the common development mistakes can be prevented.

Program Slicing is a code analysis technique that investigates program code. In Static Slicing, the code syntax can be analysed regardless of the code execution, which helps in isolating a piece of the code and investigating it further.

The measurement of robustness is still in the early stages of research. Program robustness has been reviewed and shows that the robustness measurement mostly discussed were regarding input/output validity.

In this research, program robustness is going to be measured using the program code syntax. Therefore, there should be some rules that measure the code syntax robustness. The MISRA C2 has been chosen to be these rules because they are suitable for the purpose.

The code syntax needs to be analysed to see the effect of each piece of code on the rest of the program, and the best program analysis for this research is Program Slicing. Figure 2.1 shows the top to bottom story of this research.

Chapter Three

Robustness Grid

3.1 Introduction

This chapter introduces the Robustness Grid for programs written in the C language. Measuring software Robustness needs to examine the features of programming languages in order to produce a relative scale for functions, methods, and the entire program. The Robustness Grid will show the Robustness Degree in details for a selected program.

The Robustness Grid Measurement is the process by which relative numbers are assigned to Robustness Degree of a C program in such a way to describe them according to MISRA C2 rules and their Weights.

Figure 3.1 shows the Robustness Grid building process. The Language Features and Category are the start point of the Robustness Degree measurement process. The

Language Features are the language characteristics that must be included in a program categorised process. The MISRA rules are used to assign the Language Features into categories.

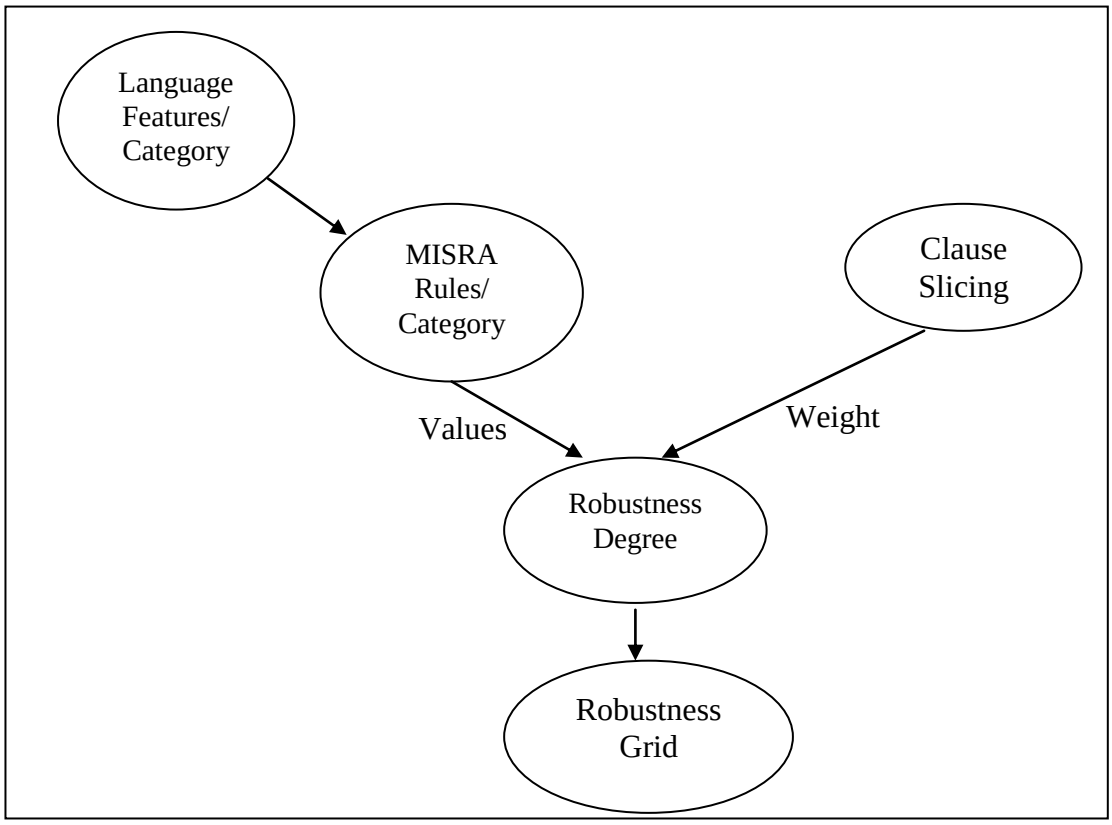

Figure 3.1 Robustness Grid Construction process

The Clause Slicing technique is used to weight the MISRA rules in order to differentiate between them in terms of rule importance and effective use throughout the program. The Robustness Degree consists of the calculations that show each Rule Robustness Degree and Function Robustness Degree using the values and weight take from the MISRA rules and Clause Slicing.

The Robustness Grid is the table by which relative numbers are assigned to Robustness Degree of a C program in a table to describe them according to MISRA C2 rules and their Weights from Clause Slicing.

The Robustness Grid shows the Robustness Degree details for each rule, function, and category, and for the whole program. It also shows the relation between functions and categories.

The program to be assessed through the Robustness Grid must fulfil the **Program Selection Criteria** which means that the program should go through the gcc compiler without any reported errors. All warnings are ignored. The Warnings are ignored because the Robustness Grid reinforces the message of the warnings.

3.2 Language Features

Language Features look at program code using different points of view; the style of the program code the use of C language standards library of the variables in the program.

Language Features are used to measure the program Robustness. Language Features are language characteristics that affect software Robustness such as arithmetic conversion, data type definition, and code control flow.

3.2.1 Language Features Categorisation

Language Features analysis checks every statement in the program and sees whether it satisfies these Features. Language Features are divided into categories depending on their shared characteristics.

Table 3.1 shows a set of Language Features as categories in MISRA C2 (MISRA 2004):

Language Features
Character Set, and Identifiers
Data types, Declarations, and Definitions
Constants
Initialisation
Arithmetic type conversions, and Pointer type conversion
Expressions
Control statements expressions, Control flow, and switch statements
Function structure
Pointers, arrays, structures, and unions
Pre-processing directives, and standards libraries.

Table 3.1 MISRA rules topics

3.2.2 MISRA Rules

3.2.2.1 MISRA rule selection

The Language Features selection mechanism from the MISRA C2 rules depended on certain conditions, and only rules that satisfy all these conditions will be part of the Robustness Grid.

The selection process for the **Robustness Language Features Conditions** from the MISRA C2 rules are as follows:

1- A rule that causes a compile time error when it is violated will be eliminated because any program that breaks such a rule does not satisfy the Program Selection Criteria condition and will not be measured in the Robustness Grid.

2- A rule must relate to the program code, and not to the environment or the documentation, because the Robustness Grid is only concerned with the program code. By this condition, the MISRA *Environment* and *Documentation*

rules categories are not selected to be a part of the language Features of the Robustness Grid.

3- The *Language extensions* rules, such as comments are not selected to be in the Robustness Grid. Only the executable code is going to be measured.

4- A rule should be easily measured. For example, rule 14.1 in MISRA C2: “there shall be no unreachable code” cannot be easily identified in general. The Clause Slicing technique has to slice all the program statements so the all the program should be executable.

In the Robustness Grid, There is no need to address the MISRA C2 rules that cause compile-time error if they breached, since these errors are caught by the compiler. After analysis, 100 out of 142 MISRA C2 rules were approved by the Robustness Features criteria and selected to be the Language Features. Not all rules will be applicable to each program.

The MISRA C2 rules are copy right protected and are thus not allowed to be published this thesis. Thus no details of the rules are given.

3.2.2.2 Robustness Features Categorisation

In this research, the Language Features are divided into 6 different categories. Each Category has a set of rules that share the same characteristics. The categories are numbered for convenience and are not intended to show a hierarchy.

These numbers indicate the order in which the categories are shown in the Robustness Grid. The Categories in the Robustness Grid must have the following characteristics:

1. Each rule in each category must be one of the MISRA C2 rules.
2. All rules in each category must satisfy the “Robustness Language Features Conditions”.

3. Each rule must be in one and only one category.

The Language Features are divided into the Categories listed in Table 3.2.

Category	Constructs	Number of MISRA C2 rules
1	The rules that consider mainly type definition, and arithmetic statements.	20
2	Rules that consider control statements, for example: if, for, and while statements.	27
3	Rules that consider function's structure.	16
4	The rules that consider arrays, pointers, and data structure such as union, struct, and enum.	4
5	The rule that consider header files and the pre-processor.	22
6	All MISRA C2 rules that were highlighted as advisory rules.	11

Table 3.2 Categories Construction

Figure 3.2 shows the MISRA rules distribution percentage for the categories. Category 2 has the largest number of rules, since it is dealing with frequent issues in the C programming language.

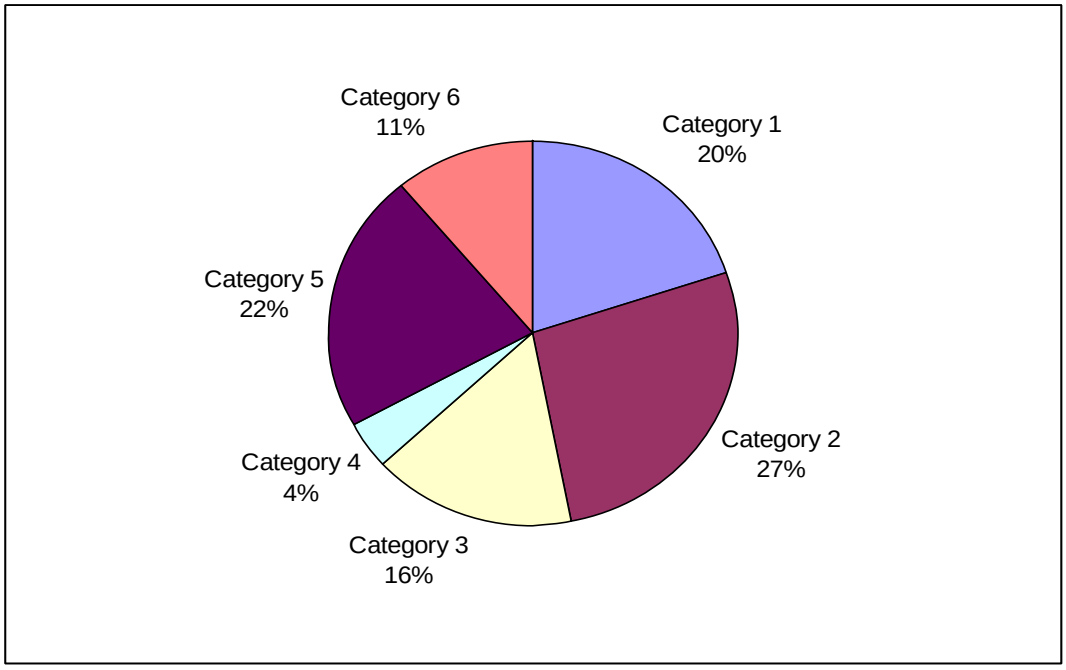

Figure 3.2 Robustness Grid Categories distributed scale

The table of all categories with their rules from MISRA C2 is shown in Appendix A.

Because the categories are independent then each Category has its own rules and these rules have no effect on each other. For example, if a program scored a low Robustness Degree in a Category, it does not necessarily means it is going to score a low Robustness Degree in other Categories. However, the Categories are still connected with each other in some Language Features Calculations to measure the final Robustness Degrees for the function or the program.

Rules in a category are not in a numeric sequence, since they are grouped depending on rule characteristics. However, rules in the same category are in numeric order according to their MISRA C2 number. The rules in a category are in X.y numbering format, where X is group number in MISRA C2 grouping program, and y is the group internal number of the rule. Thus, rule order is meaningless and does not affect the rule role in the Robustness Grid. Table 3.3 shows an example of the rules distribution into categories.

Categories	Rule Number
Category 1	4.1 + 7.1
	4.2
	5.1
	5.2
	6.1
Category 2	12.2
	12.3
	13.4
	13.5
	13.6
	14.7
	17.1
	17.5

Table 3.3 Example of rule categorisation

3.3 Clause Slicing

Slicing is a program analysis technique that allows the focus to be on the program code that is related to certain Slicing Criteria. Chapter Two (section 2.4) has reviewed a number of slicing techniques. In this research, the Clause Slicing technique is introduced.

Clause Slicing is a new slicing technique that is introduced and defined in this thesis for the first time. Clause Slicing is introduced to facilitate the Robustness Measurement of a C program.

A Clause is the minimum piece of code that can be sliced. Not every Clause is sliceable and there are some Clauses such as *#include* that cannot be sliced and this type is called the Un-sliceable Clauses. The Slicing Criteria for the Clause Slicing is **<C, n>,** where **C** is the clause, and **n** is the clause number. The Clause Slicing (C^n) is all clauses in the program that depends on C^n.

Each one of the program Clauses will be measured by the Language Features. The Clause in the programs can be defined as follows:

<break-statement>, <continue-statement>, <goto-statement> are not Clauses.

<Type-variable name> has one Clause.

<expression> '=' <expression> has the sum of the number of clauses in the both <expression>

<compound-statement>

::= '{' <declaration-list> <statement-list> '}' has one clause.

<return-statement>

::= 'return' <expression> ';' has one clause.

<do-statement>

::= 'do' <statement> 'while' '('<expression> ')' ';' has the number of clauses in the <statement> plus the number of clauses in the <expression>

<for-statement>

::= 'for' '('<initialization-expression> ';'<control-expression> ';' <iteration-expression>')' <statement> has the number of clauses in the <initialization-expression>, <control-expression>, and <iteration-expression> plus the number of clauses in the <statement>

<if-statement>

::= 'if' '(' <expression> ')' <statement> has the number of clauses in the <statement> plus the number of clauses in the <expression>.

<if-else-statement>

::= 'if' '(' <expression> ')' <statement> 'else' <statement> has the number of clauses in the both <statement> plus the number of clauses in the <expression>.

<while-statement>

::= 'while' '(' <expression> ')' <statement> has the number of clauses in the <statement> plus the number of clauses in the <expression>.

<switch-statement>

::= 'switch' '('<expression> ')' '{'<declaration-list><statement-list> <case-list> '}' has the number of clauses in the <expression> plus the number of

clauses in the <declaration-list> plus the number of clauses in the <statement-list>.

<Type-function name> '('<parameters-set>')' '{'<statements>'}' has one clause <Type-function name>, added to the number of clauses in '('<parameters-set>')' which is equal to number of parameters, plus the number of clauses in the <statement>

Introducing the Clause term to Static Slicing has brought a new type of Static Slicing: namely Clause Slicing. Clause Slice can be a Forward Clause Slicing, Backward Clause Slicing or Decomposition Clause Slicing. In this research, the interested focuses on Forward Clause Slicing, and from now on, the Clause Slicing will mean Forward Clause Slicing.

Clause Slicing is using the Clause (C) and Clause number (n) as the Slicing Criteria. A line of Code may have more than one Clause. A Clause Slice (C^n) contains all the Clauses that depend on (C^n). Figures 3.3, 3.4, and 3.5 show the differences between the Forward Slicing and Forward Clause Slicing.

Figure 3.3 shows a program with two type of numbering: the line numbers, which is the number for each line of code and used in Forward Slicing, the Clause number, which is written superscript format and used for Clause Slicing.

Clause Slicing will be used to weight the robustness Degree for program Clauses, functions, and the whole program. Each Clause in the program will be the Slicing Criteria for the Clause Slice. Thus, each Clause is going to be sliced using the Clause Slicing technique. The number of Clauses in the produced slice will be the Clause's Slice Size.

```
1. #include <stdio.h>1
2. static add (int,int);2
3. int  n3;
4. void main()4{
5.    int i = 15;
6.    int sum = 06;
7.    while (i<11) {7
8.        sum8 = add9(sum10, i11);
9.        i12 = add13(i14, 115);}
10.    printf16("sum = %d\n"17, sum18);
11.    printf19("i = %d\n"2o, i21);}
12.   static int add22(int a23, int b24){
13.       return(a+b)25; }
```

The Program has 13 lines and 25 Clauses

Figure 3.3 Gemma.c program (GrammaTech 2012)

```
6.   int sum = 0;

8.     sum = add(sum, i);

10. printf("sum = %d\n", sum);

12. static int add(int a, int b){
13. return(a+b); }
```

*Forward Slicing on (sum,6) has produced **5 lines**.*

Figure 3.4 Forward Slice on (sum,6)

```
   int sum = 06;

     sum8 = add9(sum10, );

  (, sum18);

 static int add22(int a23, ){
     return(a+b)25; }
```

*Clause Slicing on (C6{int sum=0}) has produced **8 Clauses**.*

Figure 3.5 Clause Slice on C^6=(sum=0, 6)

The use of Clause Slicing is a new idea applied to measure the importance of each clause individually. This will affect the Robustness Degree measurement for the program clauses. A clause with high influence in the program will be more important, and will have more credit in the program Robustness Degree measurement.

Other Slicing techniques such as Backward Slicing and forward are looking at what the variable is depends on, which show only the effect of the variable in the program. In Clause Slice each piece of code has been considered with its effect in the program, which makes it easier to see and measure the effect of each part in the code syntax.

3.4 Robustness Degree Calculations

The IEEE definition of Robustness is "*The Degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions*" (IEEE 1990). The emphasis in the thesis is on the latter, to measure the Robustness aspect of stressful environment conditions. Therefore, any Robustness measurement technique must generate a scale value after any Robustness measurement process.

The Robustness Degree is a relative value that shows how a clauses, functions, or the entire program is satisfy the Language Features.

The Robustness Grid calculates the Robustness Degrees for each part of the program. For each program, several Robustness Degrees are formed, as well as for each function. The Robustness Degree shows the weak and the robust points of the program, and it illustrates the changes that can be made to the program to improve its Robustness Degree.

3.4.1 Language Features Weighting

The Language Features Weight considers the role of a Clause in a program. A Clause here means the smallest piece of code that can have a slice. The un-sliceable Clauses are the *pre-processor, function prototype, typedef* and any other code that cannot be sliced. In that case, the Clause itself is considered as the Clause's slice.

The Slice Size shows the Clause and Language Features influence on the program. When the Slice Size is larger, the influence is greater.

The weighted Language Features show the Clause and rule importance in the program by the size of slice for each of them.

The Language Features are related to the Robustness Degree of code syntax and coding style. On the other hand, the Language Features Weights are related to the program Clauses and are used as a addition to the Language Features in the Robustness measurement process.

3.4.2 Data for Program Analysis

The Robustness Grid measures a program with respect to Language Features and their Weights, and shows the program robustness measurement results as numbers and percentages.

In this section, the data for program analysis in the Robustness Grid will be described in detail.

3.4.2.1 Clause Table

The Clause Table, as Table 3.4 presents, contains six columns, where each column deals with one set of program analysis data.

Clause Number	Slice Size	Clause Frequency	Clause Weight	Function Name	Applicable Rules	
					Satisfied	Violated

Table 3.4 Clause Table

The Data that is used in the Robustness Grid is produced there after the following steps:

1- All sliceable Clauses in the program will be sliced. The un-sliceable Clauses will have the Clause itself as the slice.

2- **Clause Number** shows the Clause sequence number in the program. Each Clause Number represents one Clause in the program. They are the basis of all other columns since Language Features and Language Features Weight are based on the Clauses.

3- From the Clause Slice the following can be determined:

   a. **Slice Size**: the number of Clauses in the slice. Un-sliceable Clauses will have Slice Size = 1.

   b. **Clause Frequency**: the number of times a Clause has been part of a Clause slice. The Clause frequency for un-sliceable Clauses will be one, since it only occurs once in its own slice.

   c. **Clause Weight**: The multiplication of the **Clause Frequency** by the **Slice Size**, which reflects the importance of the Clause in the program, where it shows the influence of the clause and how it affects the robustness of the program.

4- Each Clause will be compared against the MISRA C2 rules, to find out which Clause is assigned to which rule, and this will be used to create the Data Table.

5- **Applicable Rules** show whether the Language Features are Satisfied or Violated for each Clause in the program. The Applicable rules are the MISRA C2 rules that apply to each Clause of a program. Any number of rules can be applicable to a Clause. The two sub-columns that form the Applicable Rules column are: Satisfied, which shows the rules that have been satisfied, and Violated, which shows the rules that have been violated by the Clause.

6- For each MISRA C2 rule, the number of times the applicable rule is

   - Satisfied, and

- Violated

will be counted.

7- **Function Name** is the name of each function in a program. Function Name declares in which function the Clause belongs. Clauses that are not included in any function will be added to the main function.

   Function prototype and function declaration are function components in a program. These Clauses are counted as part of the function even if they are written inside another function.

The Clause Slice Size and Clause Frequency are independent, so a Clause could occur frequently in many slices while it may have a small Slice Size. This Clause can have the same weight as a Clause in the opposite position. For example, suppose Clause 5 occurred in 3 slices, which make the Clause Frequency value equal to 3, and suppose it has 8 Clauses in its slice, so the Slice Size is equal to 8. Thus, Clause 5 Clause Weight will be $3 \times 8 = 24$. Suppose another Clause, Clause 9, has the opposite situation where its Clause Frequency equals eight and Slice Size equals three. Clause 9 Weight will be equal to the Clause Weight of Clause 5 which is 24. Consider a third Clause, Clause 12. With 6 as Clause Frequency value and 4 as Slice Size, the Clause Weight will also be the same, which is 24.

This means that the Clause Weight is a factor that measures the effect and the usage of a Clause in a program. In other words, it measures the role a Clause plays throughout a program, which reflects how important this Clause is in the program.

The Clause with the biggest Weight will be the most important Clause in a program, and the level of importance for a Clause is reduced when the Clause Weight value decreases.

Table 3.5 shows a small part of the Clause Table for the SwapoAdd.c program. Appendix C is an example of how the Clause table is going to be presented.

Clause number	Slice Size	Clause Frequency	Clause weight	Function name	Applicable rules	
					Satisfied	Violated
1*	1	1	1	main	19.1, 19.2,20.9, 20.2,20.1	0
2*	1	1	1	main	19.6	0
3*	1	1	1	main	19.6	0
4*	1	1	1	main	0	6.3
5	4	2	8	main	5.1, 6.1, 8.12, 13.1, 9.2	0
6*	1	1	1	incr	16.3, 16.4, 16.1	19.7
7*	1	1	1	swap	16.3, 16.4, 16.1	19.7
8*	1	1	1	one	16.3, 16.4, 16.1	19.7
9	35	2	70	main	14.7, 16.1, 8.2, 8.6	16.5, 19.7, 8.1

Table 3.5 Example of Clause Table

3.4.2.2 Data Table

The Data Table columns, as shown in Table 3.6, are all about the rule data that apply to a program. The Data Table is strongly linked with the Clause Table since all values are taken from the Clause Table.

Rule Number	Number of Satisfied	∑Satisfied Slices	Number of Violated	∑Violated Slices

Table 3.6 Data Table

1- **Rule Number** is the rule index number in the MISRA C2 documents. This rule number is used to refer to the rule that was applied in a program for a Clause.

2- The Applicable Rule is either Satisfied or Violated. The Applicable Rule column is divided into two sub-columns: **Number of Satisfied** and **Number of Violated**. The Number of Satisfied sub-column shows how many times a rule has been applicable and satisfied throughout the whole program where the rule was counted. It presented by the sign (+) followed by a number (n) that indicate how many times a rule has been satisfied. Number of Violated

sub-column shows how many times a rule has been applicable and violated throughout the whole program. It presented by the sign (-) followed by a number (n) that indicate how many times a rule has been violated. If a rule is not applicable in the program, the Applicable Rule cell in the table will be filled with (0). The values for these sub-columns are drawn from Satisfied and Violated sub-column in the Clause Table (Table 3.4).

3- The ∑**Satisfied Slices** is a combination of two columns in the Clause Table. These columns are the Satisfied Rule and the Slice Size of a Clause. Satisfied Rule will be counted, and each time a rule is satisfied, the total of all slice sizes of the Clauses that satisfy a rule will be the ∑Satisfied Slice. This column reflects the effect of a rule throughout a program. Simply, each time a rule is satisfied, the Clause Slice Size will be added to the ∑Satisfied Slice rule value and the final value will be the ∑Satisfied Slice in the Data Table.

4- ∑**Violated Slices** column follows the same procedure as ∑ Satisfied Slices Size. The number of times a rule was violated will be counted, registered and used to calculate the Violated Slices.

Table 3.7 shows a small part of a Data Table that was created for the SwapoAdd.c program. See Appendix D as an example of how the Data Table is going to be presented.

Rule Number	Number of Satisfied	∑ Satisfied Slices	Number of Violated	∑Violated Slices
4.1 + 7.1	5	9	3	15
4.2	5	9	0	0
5.1	13	70	0	0
5.2	0	0	3	13
6.1	1	4	0	0
6.3	0	0	1	1
8.1	3	22	1	35
8.2	4	57	0	0
8.3	5	19	0	0

Table 3.7 Data Table Example

3.4.3 Rule Weighting

The Language Feature Weight aims to measure the Language Features importance level by giving each Language Feature a value that expresses its importance level. The Rule Weight Calculations is shown in Table 3.8.

Function Name				
Satisfied Weight	Violated Weight	Rule Function Frequency	Rule ∑ Function Slice Size	Rule Function Weight

Table 3.8 Rule Weight Calculations

1- **Function Name** is the name of each function in the program. The sub-columns that come under the Function Name are the calculations that are related to every function in the program. Some of these sub-columns take their data directly from Data Table (Table 3.6) in Section 3.4.2.2, or indirectly from the Clause Table (Table 3.4) in Section 3.4.2.1.

2- **Satisfied Weight** is the Number of Satisfied in the Data Table multiplied by the ∑ Satisfied Slices from the same Table. The Satisfied Weight illustrates relatively the rule satisfaction Degree between other rules.

   The Satisfied Weight is one of the main factors that reflect the function Robustness Degree. The Satisfied Weight is the Frequency of the applicable rule that was satisfied in a program multiplied by the effect of this rule in the program.

3- **Violated Weight** column is the Number of Violated rules in the Data Table multiplied by the ∑ Violated Slices from the Data Table.

   Comparing the Satisfied Weight and the Violated Weight will give an indication of the Robustness Degree of a program, and whether it has a major robustness defect or not. In addition, it will reflect the defect side of each rule in each function.

4- **Function Frequency** column is the result of adding the Numbers of Satisfied and Number of Violated columns from the Data Table. Rule Function Frequency is the number of times a rule was applicable throughout a function. It is equal to the number of satisfied rules plus the number of violated rules in the Applicable Rule column of the Data Table. This value shows which function has applied a rule the most, or least, to help the developer identify the biggest or smallest effect of a rule in the function.

5- **Rule ∑ Function Slice Size** is the total number of Satisfied and Violated Slices Size in Data Table. The value indicates the behaviour of the Applicable Rule in a function and throughout the whole program. Rule ∑Function Slice Size shows the function and Clauses that have a large effect in the program.

6- **Rule Function Weight** is the accumulative weight of the rule for the function. The Rule Function Weight is one of the main Features that are powerfully related to the Language Features where it is used to measure the Robustness Degree. The Rule Function Weight shows the influence of the rule and how it affects the program robustness as overall.

   This value is a result of multiplying the Rule Function Frequency in the Rule Weight Calculations Table, which is also as accumulative value of the number of times a rule was applied, by the Rule ∑Function Slice Size, which is the accumulative Slice Size of all the times a rule was applied. Rule Function Weight represents the Rule Weight for the whole function.

   Note that the Rule Function Weight is not equal to the addition of Rule Satisfied Weight to Rule Violated Weight.

Table 3.9 shows a small part of Rule Weight Calculations that was created for the SwapoAdd.c program as an example of how the data is going to be presented.

incr				
Satisfied Weight	Violated Weight	Function Frequency	Rule ∑Function Slice Size	Rule Function Weight
45	45	8	24	192
45	0	5	9	45
910	0	13	70	910
0	39	3	13	39
4	0	1	4	4
0	1	1	1	1
66	35	4	57	228
228	0	4	57	228
95	0	5	19	95

Table 3.9 Rule Weight Calculations Table Example

The Rule Weight Calculations Table shows the calculations that are related to the Language Features. This Table will be repeated for each function in the program, as part of the Robustness Degree measurement process. The Rule Weight Calculations Table is used to compare between functions to find the function with the largest Applicable Rules, and the function with the largest Rule Weight value. The Rule Weight Calculation Table gives a hint about the most used function and rule in the program.

3.4.4 Function Category Degree

Function Category Degree (FCD) is the Robustness Degree that a function scores in a Category. The Function Category Degree has two values depending on rule satisfaction status: **Function Category Satisfy Degree** (FCSD), **and Function Category Violate Degree** (FCVD).

FCSD reflects the satisfaction of the Function Language Features in a Category. The value is calculated by the sum of all the times all the rules in a Category were satisfied (the Total of Satisfied Rules Frequency) divided by the sum of all times all rules in a Category were applicable, presented as percentage. On the other hand, FCVD reflects the Violation of the Function Language Features in a Category. The value is calculated by the sum of all the times all the rules in a Category that were

violated (the Total of Violated Rules Frequency) divided by all times all rules in a Category have been applicable presented as percentage. FCSD and FCVD values together represent the FCD of a function for the Category in the Robustness Grid and show the function performance in terms of Robustness Language Features.

The FCSD and FCVD values are the Robustness Degree measurement of the function. They also help decide whether the function needs to be re-engineered to improve its Robustness Degree, by highlighting the defective part of the function. The Rule Weight Table with FCSD and FCVD together make the Function Robustness Grid, which is the main block of the Robustness Grid.

Categories	Rule Number	Function name								FCD %	
		Applied rules	∑Satisfied Slice Sizes	Rule Satisfied Weight	∑Violated Slice Sizes	Violated Weight	Function Frequency	Rule ∑Function Slice Sizes	Rule Function Weight		
		+n, -n, or 0 for each rule	∑all Slice Sizes of all times a rule been satisfied	=∑+n multiply by ∑satisfied Slice Sizes	∑all Slice Sizes of all times a rule been violated	=∑|-n| multiply by ∑violated Slice Sizes	|applied rule value|	∑satisfied Slice Sizes + ∑violated Slice Sizes	applied rule ∑Slice Size (∑satisfied + ∑violated)multiply by rule function frequency	FCSD % for satisfied rules	FCVD % for violated rules

Table 3.10 Function Robustness Grid with sketch equations

Table 3.10 shows the equations used to create each column in the Function Robustness Grid Table, which assess the Function Robustness Degree. Table 3.11 shows a small part of Function Robustness Grid Table that was created for function *incr* in the SwapoAdd.c program as an example of how the data is going to be presented.

Categories	Rule Number	Incr								FCD %	
		Applied rules	∑Satisfied Slice Sizes	Rule Satisfied Weight	∑Violated Slice Sizes	Rule Violated Weight	Rule Function Frequency	Rule ∑Function Slice Sizes	Rule Function Weight	FCSD %	FCVD %
Category 1	4.1 + 7.1	0	0	0	0	0	0	0	0	2/2 = 100%	0/2=0%
	4.2	0	0	0	0	0	0	0	0		
	5.1	+2	6	12	0	0	2	6	12		
	5.2	0	0	0	0	0	0	0	0		
	6.1	0	0	0	0	0	0	0	0		
Category 2	12.2	+1	2	2	0	0	1	3	3	1/3 = 33.3%	2/3 = 66.7
	12.3	0	0	0	0	0	0	0	0		
	13.4	0	0	0	0	0	0	0	0		
	13.5	0	0	0	0	0	0	0	0		
	13.6	0	0	0	0	0	0	0	0		
	14.7	-1	0	0	5	5	1	5	5		
	17.1	-1	0	0	2	2	1	2	2		
	17.5	0	0	0	0	0	0	0	0		

Table 3.11 Example of Function Robustness Grid

3.5 Robustness Grid

The Function Category Degree (FCD) is the part of the Robustness Grid that measures the program function. FCD will have the same structure for every function in the program but with new values related to each function.

All functions in the program have calculations to measure the program as one piece, including all function calculations in the Function Category Degree.

3.5.1 Program Category Degree

The **Program Category Degree** (PCD) is a part of the Robustness Grid that measures the whole program using all the calculations in the functions that the program has, all Language Features and all Language Features Weight that have been applicable and calculated.

3.5.1.1 Program Category Degree (PCD)

The Program Category Degree (PCD), Table 3.12, is the accumulative column for all Functions for each Category Degree in the program (∑FCD). The PCD is the Robustness Degrees in terms of Language Features for all rules that were applied in the program. Since PCD is the total of Robustness Degrees of all functions, it also has two values: **Program Category Satisfaction Degree** (PCSD), and **Program Category Violation Degree** (PCVD).

PCD%	
PCSD % for satisfied rules	PCVD % for violated rules
∑PCD for previous categories for satisfied rules	∑PCD for previous categories for violated rules

Table 3.12 Program Category Degree Table

The PCSD is the whole program Satisfaction Degree for each category. It is the sum of the number of times a rule was satisfied in all functions in a Category, divided by the sum of the number of times a rule was applicable in all functions for the same Category, presented as a percentage. The PCVD is the whole program Violation Degree for each category. It is the sum of the number of times a rule was violated for all functions in a Category divided by the sum of all times a rule was applied for all functions for the same Category, presented as percentage.

3.5.1.2 Accumulative Categories values (AC)

In the Robustness Grid, the rows show a different measurement from the columns. In the Robustness Grid rows are the Language Features and Language Features Weight measurement. Part of the Robustness Grid rows is the **Accumulative Category** (AC), which shows an accumulative value for the number of rules that have been applicable in Categories, Functions, and the whole program, and presents the accumulative rules calculations of the previous Categories. The AC value is calculated for all Language Features and Language Features Weights in the Robustness Grid which is presented in the Function Robustness Grid (Table 3.10), as well as the Robustness Degree for the entire program.

The **FAC** is the **Function Accumulative Category** value, which is the accumulative value of all categories of a function or the entire program. The AC and FAC are used to compare functions' Robustness Degrees, which help determine which function is the least or the most robust in term of Language Features and/or Language Features Weight. They also help specify which function with the most effect (Slice Size) in the program and determine its Robustness Degree.

As in all columns, PCD columns have the AC row. The AC row of PCD calculates the accumulative values of both parts of PCD through categories showing the Robustness Language Features Degrees for all functions through the Categories, and how the Robustness Degree is affected in each category.

In the Robustness Grid, the intersection of FAC row with Function Category Degree (FCD) column is the **Whole Category Function Degree** (WCFD). WCFD shows the Robustness Degree for a Function in all Categories. WCFD will also have two Degree values: Satisfaction and Violation.

Furthermore, the intersection of FAC row with PCD column is the **Whole Program Degree** (WPD). WPD presents the Language Features Satisfaction and Violation

Degree for the whole program in all Categories. The WPD illustrates the final Robustness Degree that a program scores for all Language Features that were applied in all Categories. This Degree is presented as percentage to show the relative measurement to the whole program Robustness Degree. The **Whole Satisfied Program Degree** (WPSD) and **Whole Violated Program Degree** (WPVD) are detailed the WPD of the program. The columns in Table 3.13 show the AC and FCD equations that are part of the Function Robustness Degree measurement.

	Function name								FCD %	PCD %	
AC	number of rules that have been applicable in previous categories	function ∑satisfied Slice Sizes	function satisfied weight	function ∑violated Slice Sizes	∑violated weight in a function	frequency for all rules in a Function	∑all rules Slice Size in a Function	∑all rules weight in a Function	∑FCS for previous categories for both satisfied and violated rules	∑PCD for previous categories for both satisfied and violated rules	
FAC	AC of all categories	AC of all categories	AC of all categories	AC of all categories	AC of all categories	AC of all categories	AC of all categories	AC of all categories	WCFD	WPD	
										WPSD	WPVD

Table 3.13 AC and FAC equations

3.5.2 Category Calculations

The Category Calculations part of the Robustness Grid is related to Rules Categories, where each Rule and Category will be analysed and measured using Language Features characteristics: Rule Slice Size, Frequency, and Weight.

Category Calculations measure the Language Features individually and in the Categories as well. The Language Features characteristics measure the Language Features and the Categories show how a Rule or a Category is effective in the program.

	CATEGORY CALCULATIONS								
Categories and rules number	Category Satisfied Frequency	Category Satisfied ∑Slice Sizes	Category Satisfied Weight	Category Violated Frequency	Category Violated Slice Sizes	Category Violated Weight	Category Frequency	Category ∑Slice Sizes	Rule Category Weight
	∑functions satisfied frequency	∑functions ∑satisfied Slice Sizes	∑functions frequency ∑function's satisfied frequency	∑functions violated frequency	∑functions ∑violated Slice Sizes	∑functions frequency ∑function's violated frequency	∑all functions frequency in a category	∑functions ∑ Slice Sizes	∑category frequency ∑functions violated frequency
AC	∑Category satisfied frequency for previous categories	∑Category satisfied Slice Sizes for previous categories	∑Category satisfied weight for previous categories	∑Category violated frequency for previous categories	∑Category satisfied Slice Sizes for previous categories	∑Category violated weight for previous categories	∑Category frequency for previous categories	∑Category ∑Slice Sizes for previous categories	∑Category Weight for previous categories
FAC	AC for all categories	AC for all categories	AC for all categories	AC for all categories	AC for all categories	AC for all categories	AC for all categories	AC for all categories	WPW

Table 3.14 Category Calculations

1- **Category Satisfied Frequency** is the total number of times a rule was satisfied in all functions in the program, which is equal to number of satisfied times for a rule in all program functions.

2- **Category Satisfied Slice Sizes** is the total number of all Slice Sizes for all Clauses that are applicable and satisfy a rule through all functions in the program.

3- The multiplication result of Category Satisfied Slice Sizes by Category Satisfied Frequency is the **Category Satisfied Weight**. The Category Satisfied Weight presents the Rule Weight in the whole program, which measures the rule effectiveness in the program. The rule with the highest weight value will be the most effective rule in the program. The rule with the highest Satisfied Weight value will be the most successfully effective rule in a program.

4- **Category Violated Frequency** is the number of times a rule was violated through all functions. It is the same value of number of violated times for a rule in all functions.

5- As with the Satisfied Rules, the Violated Rules will have the same measurement equations. **Category Violated Slice Sizes** is the total number of all slices for all rules where the slice was applicable and violated a rule through all functions. This is the same as the total number of all function Violation Slice Sizes.

6- **Category Violated Weight** is the result of multiplying Category Violated Slice Sizes by Category Violated Frequency. The Violated Weight reflects the rational value that specifies the defective part of the program that needs to be improved to increase the Robustness Degree. The violated rules will give a clue about things that should be changed to get a higher Robustness Degree.

The general category attitude is shown by the **Category ∑ Slice Sizes**, **Category Frequency**, and **Rule Category Weight**. These three columns show the general performance of the Language Features Categories in all program functions. They also show which Category is the one with the largest Weight,

the one with the highest frequency used, and the one with the most important rules. These features identify the importance of each category.

7- **Category Frequency** expresses the number of times a rule was applicable in all functions. It is equal to Total number of times of satisfied and violated for a rule in all functions. This value shows the number of times a rule was used in the program Clauses.

8- **Category ∑ Slice Sizes** are the total number of slices of a Clause that apply a rule in all functions. This column describes the influence of the rule inside the program by showing the size of a slice(s) that will be affected by this rule.

9- **Rule Category Weight** is the result of Category ∑ Slice Sizes multiplied by Category Frequency in Category Calculation Table, which reveals the rule importance compared with other rules regardless the Rule Satisfaction Weight or Degree.

The importance of a rule will help in maintaining the important rules to get a significant improvement of the general Robustness Degree. The rule with largest Category Weight is the rule that has the largest impact on the program Robustness Degree.

10- **Whole Program Weight** (WPW) is used in the measurement of the defect of each Rule, or Category. WPW is another way to measure the rules, the Categories, and the Functions importance and effectiveness.

11- **AC and FAC** are mentioned before in the Section 3.5.1.2.

Table 3.15 shows an example of how the Category Calculations are presented in the Robustness Grid.

CATEGORY CALCULATIONS									
Category and rule numbers	Category Satisfied Frequency	Category Satisfied ∑Slice Size	Category Satisfied Weight	Category Violated Frequency	Category Violated ∑Slice Size	Category Violated Weight	Category Frequency	Category ∑Slice Size	Rule Category Weight
	5	9	45	3	15	45	8	12	96
	5	9	45	0	0	0	5	9	45
	13	70	910	0	0	0	13	70	910
	0	0	0	3	13	39	3	13	39
	1	4	4	0	0	0	1	4	4
AC	24	92	1004	6	28	84	30	108	1094
FAC	104	634	3609	25	197	350	128	823	4610

Table 3.15 Category Calculations example

Function Calculations mentioned in section 3.4.3 look at the program functions individually. All Function Calculations depend on the function and Language Features applicable in that function. In Robustness Grid, Function Calculations measure the Rules vertically, starting from the first rule in the first Category, moving down till the last rule in the sixth category.

In Program and Category Calculations, the Calculations are more about the Language Features behaviour through the program in general, and these calculations measure the Language Features Categories. In the Robustness Grid, Program and Category Calculations measure the rules horizontally. For each rule, the measurement starts from first rule in first function in the Robustness Grid moving right till the last function.

3.6 Summary

The Robustness Grid measures the Robustness Degree of the functions and the program written in C language. The Robustness Degree is a relative value that

shows the Robustness Features satisfaction status of the function and the program. Robustness Features are characteristics that affect the program Robustness Degree. These Features are: Language Features and Language Features Weight.

Language Features are code independent, where a set of Language Features are selected, categorised, and used to measure the Robustness Degree.

The Language Feature Weights are produced by the Clause Slice technique and some mathematical equations. The Language Feature Weights depend on the program code.

The Clause Table is a table created by the Clause Slice technique, and used to calculate the Language Features Weights and the Robustness Degree. The Clause Table shows the importance of the program Clauses based on the Clauses Slice Size, Clause Frequency, and Clause Weight, where the Clause with highest weight is the most important Clause in the program.

The Robustness Grid is a table that combines both the Data Table and the Clause Table. The Robustness Gird indicates a relative scale that illustrates the weak points of the program that reduce the Robustness Degree of the Program. Pointing out the rules that have a problem helps the developers and maintainers to raise the Robustness Degree by fixing the defective Clauses that score low Robustness Degrees. The equations of the Clause Table and the Robustness Grid are shown in Appendices K and L, respectively.

Chapter Four

Implementation

4.1 Introduction

The implementation chapter shows the process steps that have been followed to produce the Robustness Grid in terms of tools; the process is not fully automated. However, some of process steps were done by tools already available in the market. The language features identified in Chapter Three are encapsulated by the use of MISRA C2 rules (reviewed in Chapter Two).

The implementation process starts with a C program, going through manual measurement using the MISRA C rules. Furthermore, the program is sliced using the CSurf tool. The Slices and MISRA C rules measurements will be weighted through calculations done and presented using an Excel sheet.

4.2 Implementation Models

The implementation process described the robustness measurement of C program to produce the Robustness Grid. The model, Figure 4.1, describe the implementation process in high level.

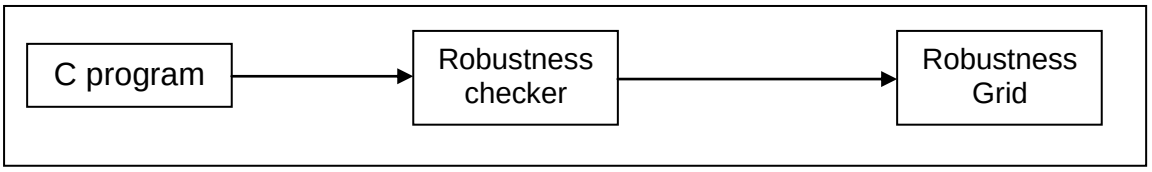

Figure 4.1 Implementation High Level model

In the Implementation High Level model, Figure 4.1, a **C program** is the start point of the implementation process. The C program will enter the **Robustness checker** to produce the **Robustness Grid**.

In the Implementation Intermediate Level, Figure 4.2, expands the Robustness Checker in the High Level model and is divided into two main pieces: **Slicer** and **Robustness Features Checker**. The Slicer is a tool that does the Clause Slicing with some related computation such as Clause Slice Size in the Clause Table (see 3.4.2.1). The Robustness Features checker takes the slicer output as an input, and generates the Robustness Grid as an output.

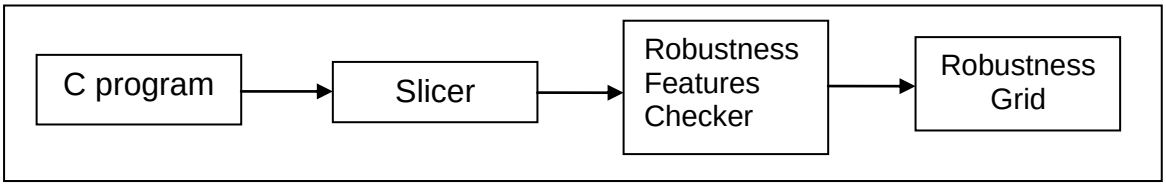

Figure 4.2 Implementation Intermediate Level model

The Implementation Low Level model, Figure 4.3, has more details of the robustness measurement procedure. A legal C program is a C program that meets the program selection criteria mentioned in Section 3.1 and is qualified to be certified by the Robustness checker.

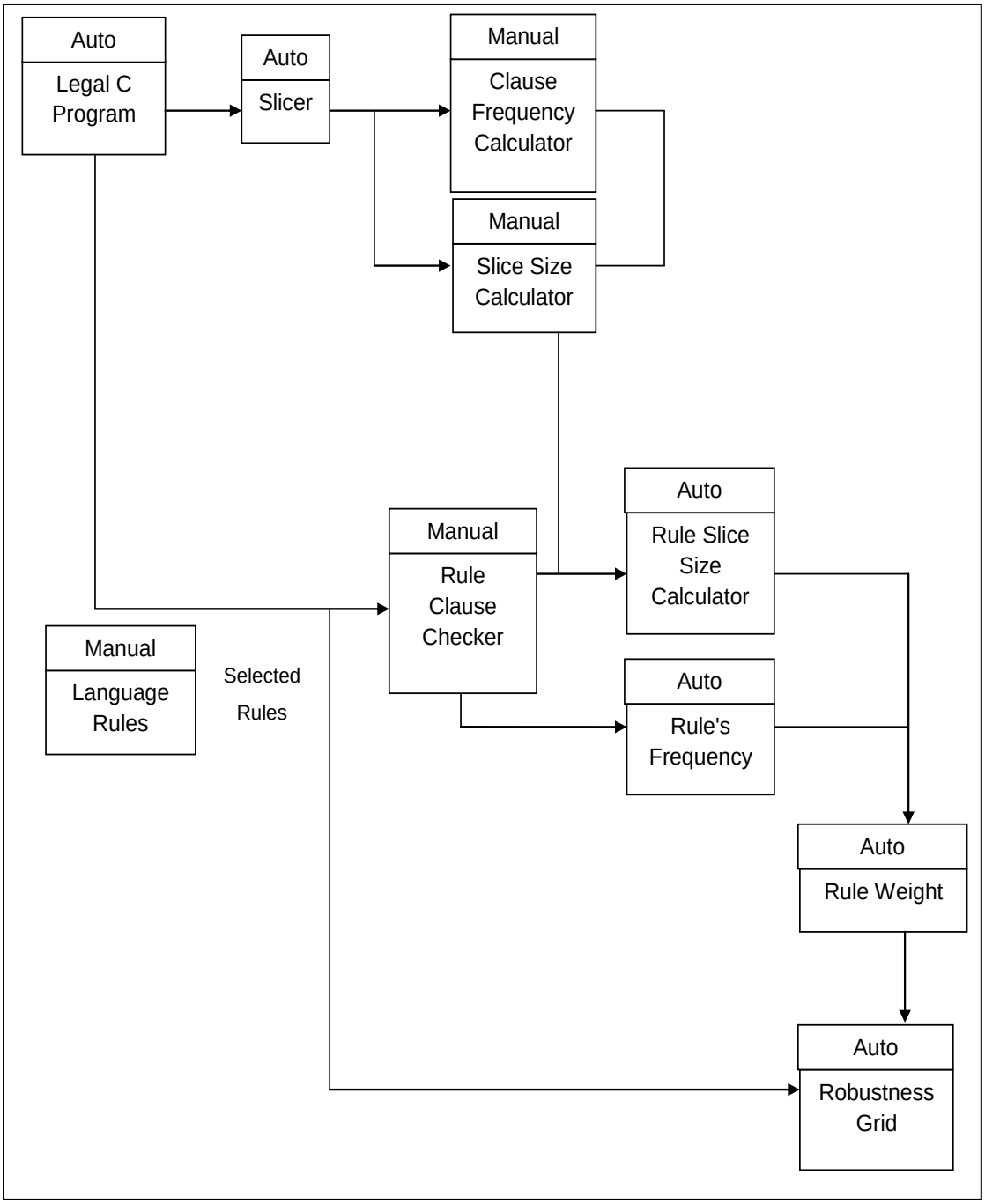

Figure 4.3 Implementation Low Level model, where *Auto* means that the part is automatically computed and *Manual* means it is manually computed.

The following table describes each entity of the low level model:

Entity	Description
Legal C Program	A C program that satisfies “program selection” conditions.
Slicer	A slicing tool.
Clause Frequency Calculator	Counting the number of times a Clause has been in a slice.
Slice Size Calculator	Counting the number of Clauses in a slice.
Language Rules	MISRA C2 guidelines.
Selected Rules	Set of MISRA C2 guidelines that satisfy the “rule selection” conditions
Rule Clause Checker	A tool that certifies a program’s Clauses using MISRA C2 guidelines.
Rule Slice Size Calculator	A calculation that counts the slice size of all Clauses that satisfy a rule.
Rule’s Frequency	Number of times a rule was applied.
Rule Weight	The importance Rule measurement which is equal to the multiplication of Rule Frequency by Rule Slice Size.
Robustness Grid	The final table containing all details, calculations, and robustness degrees.

Table 4.1 Low Level Terms

The in the Implementation process Low Level starts with a **Legal C program**, where it is used as an input for both the **Slicer** and the **Rule Clause Checker**. In the Slicer, each Clause in the program will be in the slicing criteria. The Slices enter the **Clause Frequency Calculator** and the **Slice Size Calculator** to produce the Clause Weight.

Back to the start point, the legal C program will be certified by a selected group of **Language Rules**. The certified Clauses will be combined together in the **Rule Clauses Checker** where it will produce a set of rules that were applied in each Clause with their satisfaction status. The outcome of the Rule Clause Checker and the output of the Slice Size Calculator will be joined together in the **Rule Slice Size Calculator** to produce the Slice Size for each rule.

The Rule Clause Checker will be used in the **Rule's Frequency** to find out the times a rule is being applied. By multiplying the Rule Slice Size by the Rule's Frequency, the result will be the **Rule Weight**. All of the previous entities together will construct the **Robustness Grid**.

4.3 Implementation Tools

The tools used to implement the Robustness Grid are:

1. **CSurf**®: a slicing tool used to produce the Clause Slice for each Clause in a program, which helps find a robustness measurement.
2. **MS Excel**®: a tool used to do the Function calculations, Program Category Degree, and Category Calculations. Also, it is a tool that organises the Robustness Grid to show each Function Calculations, and highlight the main calculations such as FAC, WPAC, and WPW.

4.4 Summary

The Robustness Grid contains MISRA C2 rules with their Categories, in addition to the Function Calculations, the Program Calculations, and the Category Calculations. Each one of these parts is created or measured by some tool or technique.

MISRA C2 rules are provided by the MISRA Organisation that provided these rules as C language standards. The Function Calculations are the MISRA C2 rules manual measurements for the function and with some help from CSurf for Slices and Weight, and MSExcel for doing the calculations. These tools are used to do the Program and Category Calculations as well.

The Implementation process can simply be described as follows:

1. Each Clause in the program is assessed against all the selected MISRA C2 rules.

2. All selected rules will be put in their categories depending on the Robustness Featuring Categorisation method defined in Section 3.2.2.2.
3. Each rule has the applied status next to it, showing whether it is satisfied, violated, or not applicable.
4. Program Clauses will be sliced to create the Clause Table in Section 3.4.2.1 (Table 3.4).
5. MISRA C2 Rules will be applied depending on the Program Clauses.
6. The Rules will be measured using the Clause Table to Create the Data Table in Section 3.4.2.2.
7. Program Clauses will be grouped by their function.
8. For each function, the satisfaction status, Slice Size, Frequency, and Weight of all rules is listed.
9. The Robustness Grid calculations are made for each Function (FCD) and Category (ACD) in Section 3.5.1.2 (Table 3.12), and for the entire program (PCD) Section 3.5.1.1 (Table 3.12).
10. The Category Calculations are calculated for all rules are made, detailed in Section 3.5.2 (Table 3.14).
11. The Function Calculations, Program Calculations, and Category Calculations all together create the Robustness Grid.

Chapter Five

Results

5.1 Introduction

The Robustness Grid measures the Robustness Features for programs written in the C language. This chapter will present the result of a case study of the Robustness Grid using an example program. The example is a small C program and will be used to follow each step of Robustness Grid constructing process.

5.2 SwapoAdd.c - The C program

In this case study, the SwapoAdd.c program has been written according to the Program Selection Criteria discussed in Section 3.1.

SwapoAdd.c, see Appendix B, has four functions: *main*, *swap*, *incr*, and *one*. Function *one* has one parameter and prints it every time a condition is satisfied.

Function *swap* exchanges two pointers. Function *incr incr*ements its first parameter by the value in its second parameter, and the function *main* is the main program.

5.3 SwapoAdd.c Clause Table

The Clause Table, in Appendix C, shows the program Clause characteristics: Clause Number, Slice Size, Clause Frequency, Clause Weight, Function Name, and Applicable Rules. In this Table, the first column has the Clause Number. If the Clause Number is followed by a star (*), it means this Clause is considered an un-sliceable Clause. The SwapoAdd.c program has 60 Clauses, 7 of which are un-sliceable.

The Slice Size is the number of Clauses in a slice. The Slice Size is produced by slicing on the Clause that is listed in the Clause Number column.

In the SwapoAdd.c program, Clause number 9 has the largest effect as shown in Figure 5.1, because it has the largest Slice Size. In the SwapoAdd.c program, Clause 9 is the definition of the function *main* (`int main ()`). The main function has the most effect in the program, which is always true because the main function call all function in the program, directly and indirectly.

The second most effective Clause is Clause number 27 (`int index`) with Slice Size value = 17. It is used by different Clauses, and the *for* loop depends on it because it uses it as a variable index, and thus it has more effect than other Clauses.

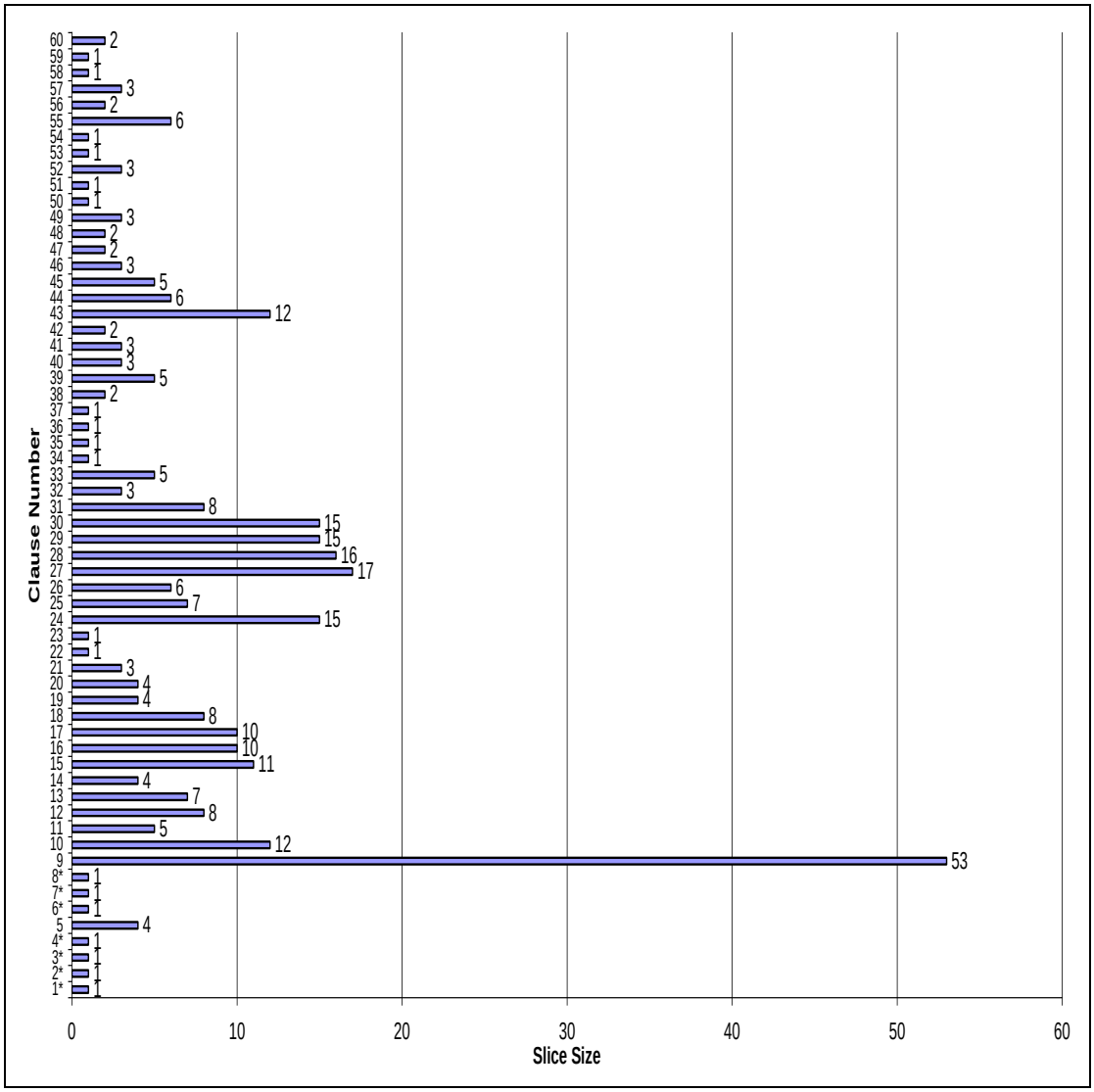

Figure 5.1 SwapoAdd.c Clause Slice Size

Another factor that measures the Clause is Clause Frequency. Clause Frequency is the number of times a Clause has been in a slice, including the Clause itself as a slice. Clause Frequency measures the use of the Clauses in the program, where the Clause with highest frequency is the one most used. Un-sliceable Clauses have a Clause Frequency equal to 1, since un-sliceable Clauses are only contained in their own slice.

In Figure 5.2, Clause number 23 has the highest Clause Frequency (15). Clause 23 is the definition of the variable *sum*, and this variable is the most used variable in the program. Clause 54 has the second largest Clause Frequency with 13, and then

each of the Clauses 42, 51, and 59 occur 12 times in a slice. However, Clause Frequency always depends on the Frequency of the Clause used.

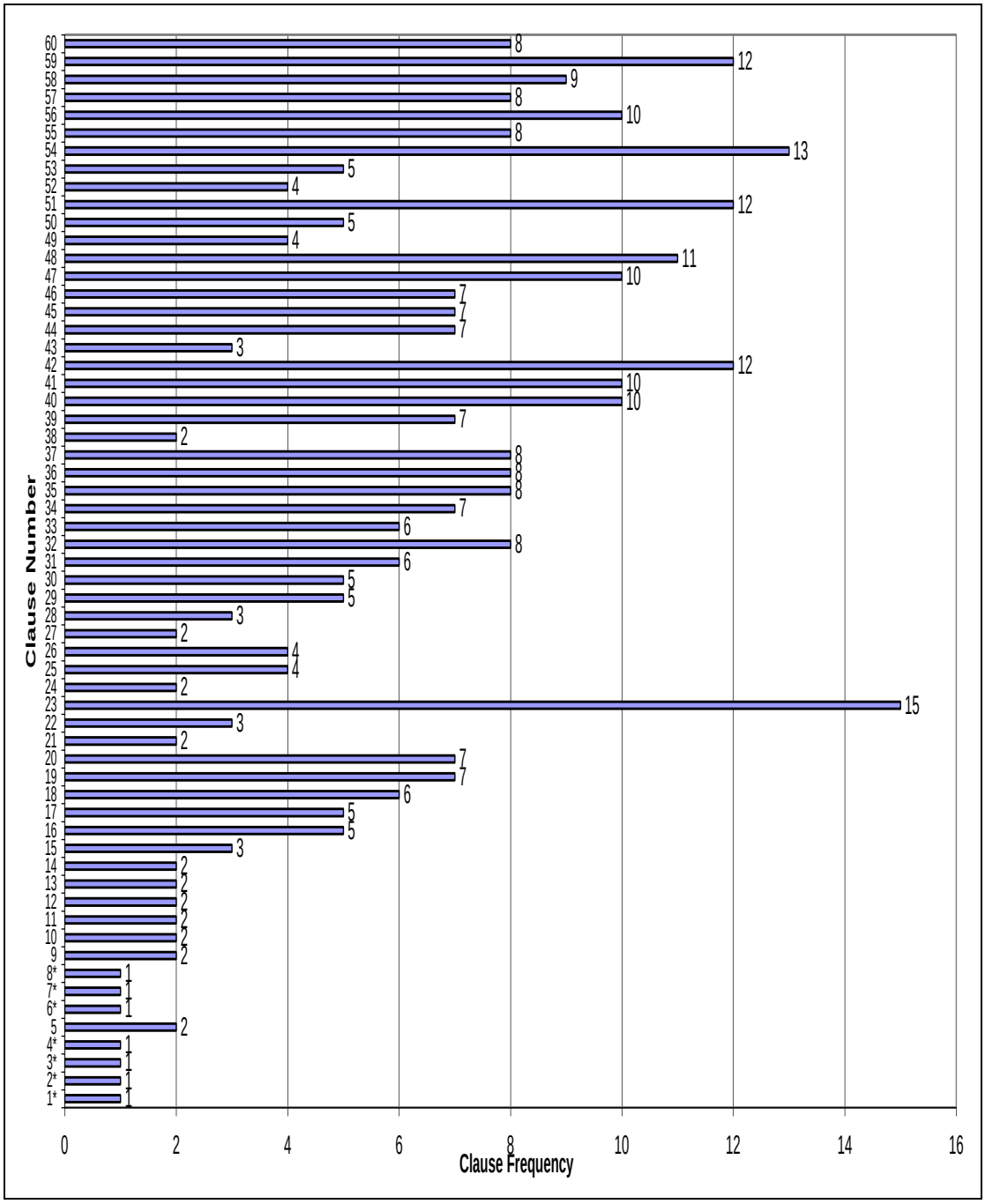

Figure 5.2 SwapoAdd.c Clause Frequency

Clause Frequency and Slice Size are independent. A Clause could have a high value for Clause Frequency but a small value for Slice Size, and vice versa.

Clause Frequency and Slice Size both generate the Clause Weight. The Clause Weight measures the Clause's importance in the program since it includes the Slice Size that measures the Clause effect and Clause Frequency which measures number of times the Clause has been used in slices.

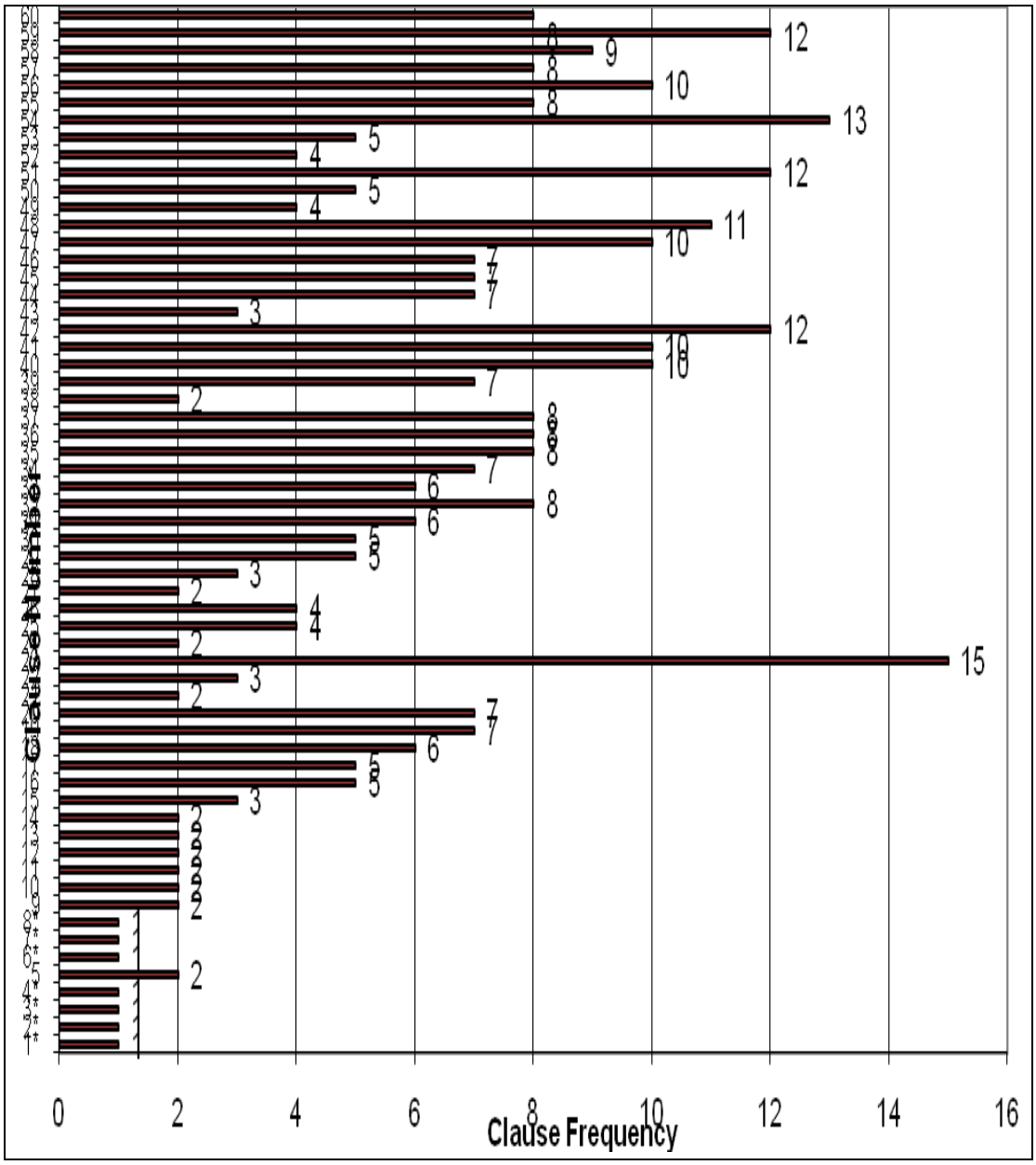

Figure 5.3 SwapoAdd.c Clause Weight

If two Clauses are equal in weight, it means that they have the same importance level. However, it does not necessarily mean that they are in the same function, Clause Frequency, or Slice Size. For example, Clauses 10, 26, 32 and 42 have the

same Weight, where Clause 10 is in *main*, Clause 26 in *swap*, Clause 32 in *one*, and Clause 42 in *incr*. Also, Clause 10 has Slice Size = 12 and Clause Frequency = 2, and by calculation, the Clause Weight is 12×2= 24. For Clause 26, the Slice Size = 6, Clause Frequency = 4, and thus the Clause Weight is 6×4= 24. In Clause 32 the Slice Size = 3, Clause Frequency = 8, which means the Clause Weight is equal to 3X8 = 24. Same as in Clause 48 which has Slice Size = 2 and Clause Frequency = 12 which means the Clause Weight is 2×12=24. This case is also repeated in other Clauses, such as 20 and 25, and in Clauses 18 and 28.

The Clause Table also shows the rules that were satisfied or violated by each Clause in the SwapoAdd.c program. The Clause Table will help the developers and maintainers by showing them which Clauses need to be maintained to improve the Robustness Degree of the Program.

5.4 SwapoAdd.c Data Table

The Data Table, in Appendix D, is the table that shows the rule characteristics: Rule Number, Number of Satisfied, ∑ Satisfied Slice, Number of Violated, and ∑ Violated Slice. The Data Table is based on the rules, whereas the Clause Table is based on the Clauses. In the Data Table, the rule status is measured through the whole program, depending on how many times the rule was satisfied or violated and how that affected the program.

The Rule Number column contains the rules that are in the Applicable Rules column in the Clause Table. Rules in the Rule Number column are sorted in numerical order using their number in the MISRA C2 definition.

The columns Number of Satisfied and Number of Violated show the number of times a rule has been satisfied or violated in the program.

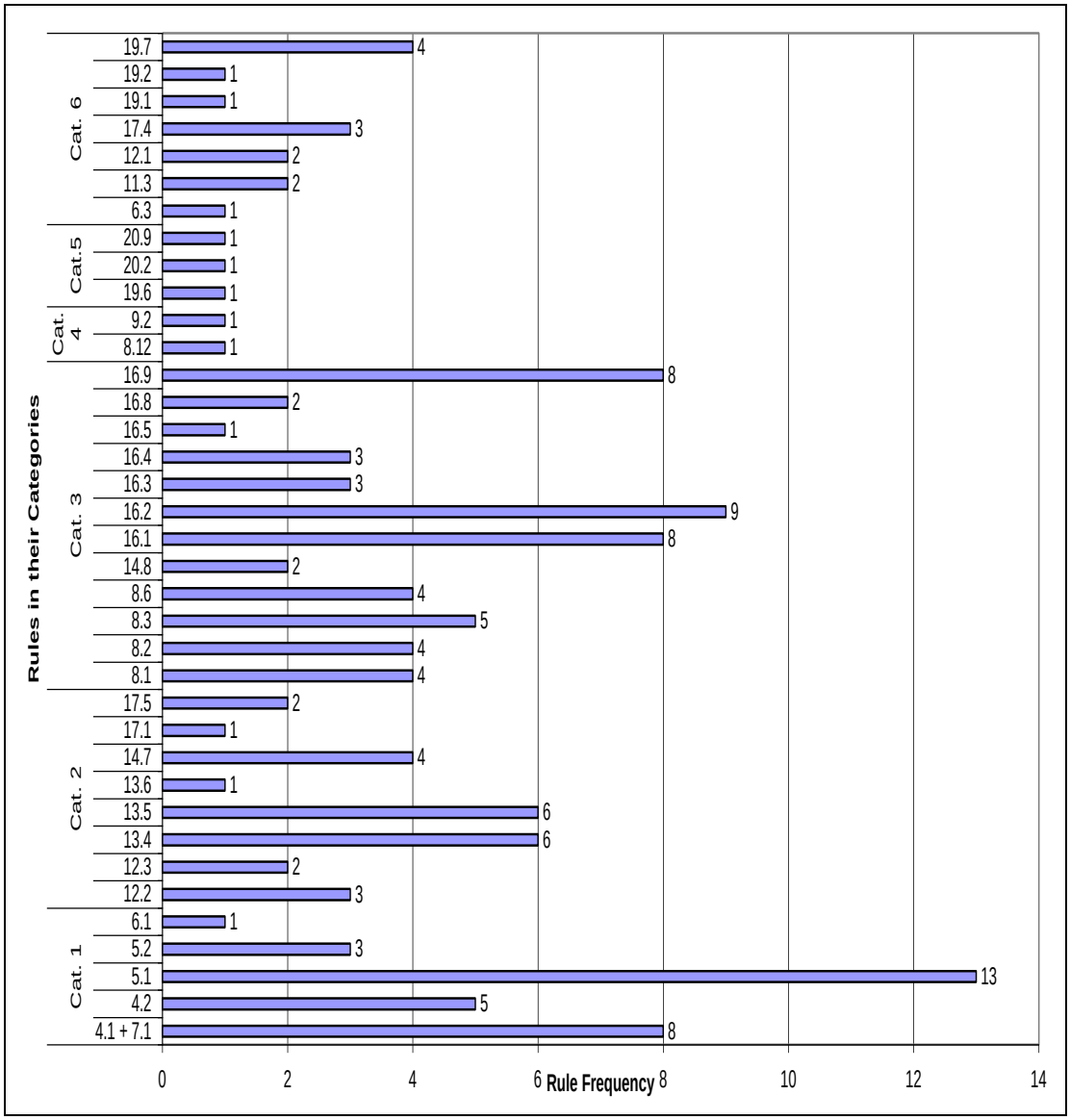

Figure 5.4 Rule Frequency

Figure 5.4 shows the Rule Frequency which is equal to the number of times a rule been applied and shown as the Category Frequency in the Category Calculation table (see Appendix J). Rule 5.1 is the most frequent rule because it considers the number of characters in identifiers. Therefore, every time an identifier is mentioned, the 5.1 rule is applied. 12 rules have the lowest number frequency with a value of 1.

The Applied Rules on the SwapoAdd.c program (see Appendix J), the rule may have one of these three scenarios: First, the rule is always satisfied in all the times it has been applicable, Rule 5.1 as an example. These rules have 0 values in the Number

of Violated and ∑ Violated Slice, and the Rule Slice Size and Frequency in the Robustness Grid is equal to the Number of Satisfied and ∑ Satisfied Slice.

Second, Rules that are always violated every time they have been applied in the program, such as Rule 19.7. These rules score 0 values in the Number of Satisfied and ∑ Satisfied Slice, and the Rule Slice Size and Frequency in the Robustness Grid is equal to the Violated Slice Size and Frequency.

Third, rules that have been satisfied in some Clauses and violated in others such as Rule 16.1. The Rule Frequency for these rules will be equal to Number of times a rule has been applied (both Satisfied and Violated). Rule Slice Size is the sum of the Total Satisfied Slices and the Total Violated Slices.

Figure 5.5 shows the satisfaction and violation relative relation of all Applicable Rules in the SwapoAdd.c Program.

There are 25 out of 37 rules are satisfied every time they were applied, and 7 out of 37 rules that are violated all the times they were applied. The remaining 5 rules are satisfied and violated in different places in the program.

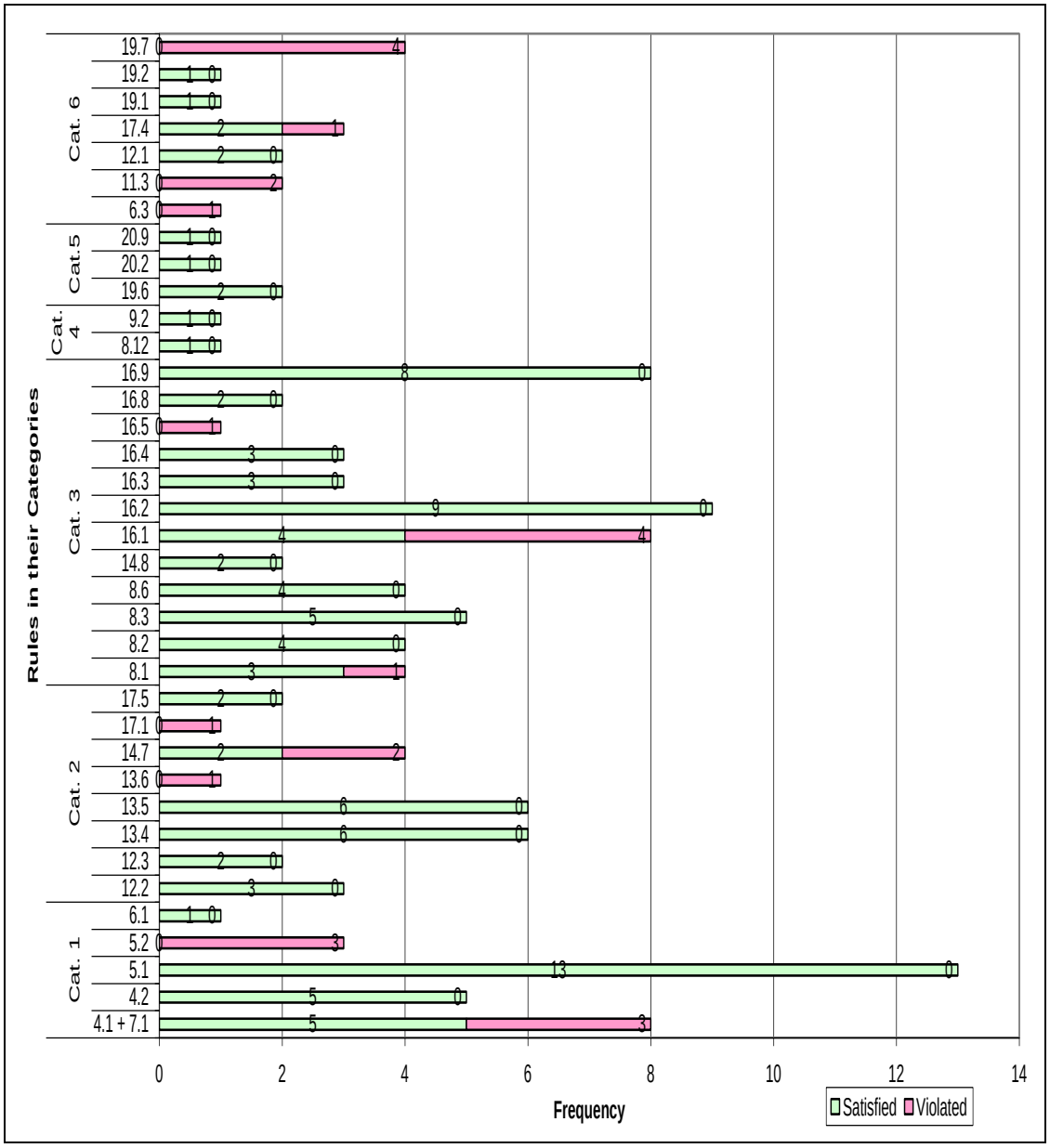

Figure 5.5 Rule Satisfaction/Violation Frequency Comparisons

5.5 SwapoAdd.c Robustness Grid

5.5.1 Functions Category Degree

In the Function Calculations, the 37 rules that are applicable will be used to measure the four functions in the SwapoAdd.c program.

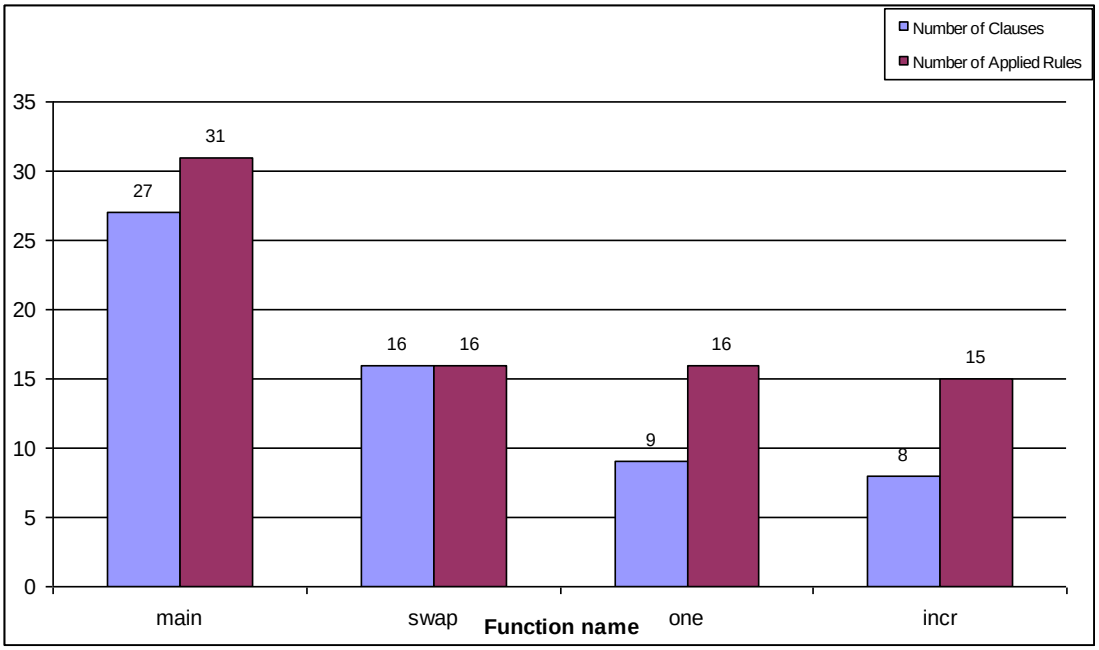

Figure 5.6 Number of Clauses and Applied Rules in each function

Figure 5.6 shows that in the SwapoAdd.c program there is a positive relationship between the number of Clauses and the number of rules have been applicable in the function. The positive relation may not be the case in another program. The function *main* has the highest number of Clauses and the highest number of Applicable Rules. Function *incr* has the lowest number of Clauses and Applicable rules in the program.

9 rules out of the 37 rules are applied in all the program functions. These rules can be used to give a general idea of how the functions have different or similar styles, and whether the developer took care of these rules during the program writing or not.

5.5.1.1 *incr* Function

Function *incr* has 8 Clauses that make it the function with the smallest number of Clauses in the program. 15 out of 37 rules are applicable in the *incr* function. These rules were applied 17 times; 13 times were satisfied and 4 times were violated. Therefore, the Function Satisfied Accumulative Categories (FSAC) for the *incr* function = (13/17) %= **76.5%**.

In Categories 1 and 3, the *incr* function has 100% Function Category Satisfaction Degree (FCSD), which means that the *incr* function satisfied all Applicable Rules in these two categories. However, the *incr* function fails to satisfy any Applicable Rules in Category 6 where the function scores 100% in Function Category Violated Degree (FCVD). In Category 2, the *incr* function satisfies 1 out of 3 Applicable Rules, which means the values of FCSD and FCVD are 33.3% and 66.7%, respectively.

In Categories 4 and 5, the function has no Applicable Rules, which leave the Robustness Degree values not applicable or equal to 0 for both of FCSD and FCVD.

Figure 5.7 shows the number of Applicable Rules in the *incr* function. It also shows the number of times the Applicable Rules have been satisfied and/or violated. Some of the Applicable Rules are applied on the program more than once, and this is why in some cases the satisfied times are more than the number of Applicable Rules. The function *incr* will be discussed in details in section 5.6.1 as a part of the result evaluation.

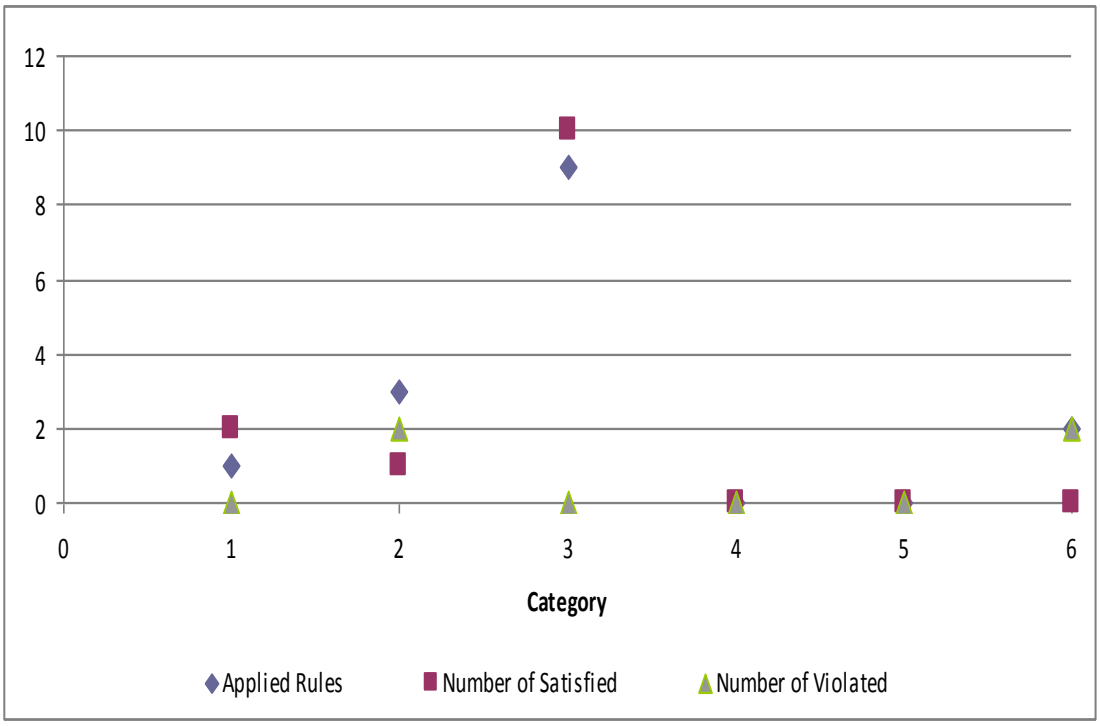

Figure 5.7 Applicable rules for function *incr*

5.5.1.2 *swap* Function

Function *swap,* as shown in Appendix E, has 16 Clauses, 16 rules out of 37 applied 29 times and they are satisfied 21 times, and violated 8 times, which means that FSAC = (21/29) % = **72.41%.**

The *swap* function fails to have 100% FCSD in any of the 6 Categories. Moreover, the *swap* function has 100% FCVD in Categories 2 and 6. This does not affect the satisfaction degree since the most applicable rules are applied in the remaining Categories. In Category 1, the *swap* function satisfies 7 out of 9 applicable rules, which means that the values of FCSD and FCVD are 77.78% and 22.22%, respectively. The satisfaction percentage rises in Category 3 to 87.50% where the rules are satisfied 14 times out of the 16 times they were applied.

For Categories 4 and 5, the function has no Applicable Rules, and this leaves the Robustness Degree values equal to 0 for both of FCSD and FCVD.

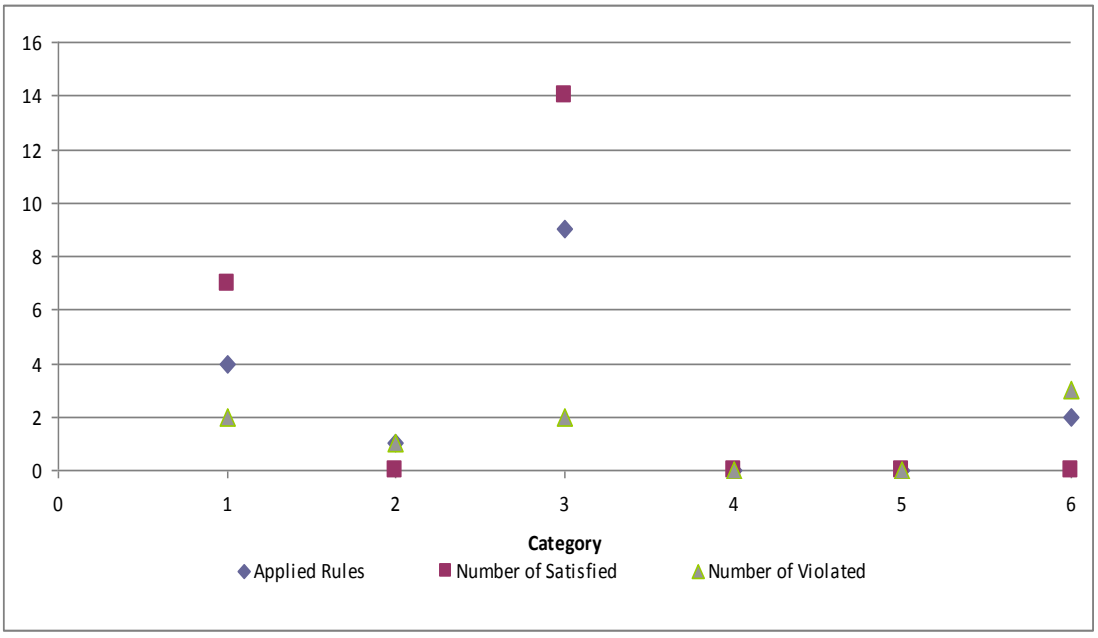

Figure 5.8 Applicable rules for function *swap*

Figure 5.8 shows the Applicable Rules in the *swap* function.

The full Function Calculation Table of function *swap* is shown in Appendix E.

5.5.1.3 *one* Function

Function *one,* as shown in Appendix F, has 9 Clauses, 16 rules out of 37 have measured the Robustness Degree and these rules have been applied 19 times; 16 times they have been satisfied and 3 times they have been violated, which mean the FSAC = (16/19) % = **84.21%.**

In Category 2, the *one* function has 100% Function Category Satisfaction Degree (FCSD), and in Categories 1 and 3, the function score a high robustness degree; 75% and 92.3%, respectively. However, the *one* function fails to satisfy any Applicable Rules in Category 6 where the function scores 100% in Function Category Violated Degree (FCVD).

In Categories 4 and 5, the function has no Applicable Rules and this leaves the Degree values equal to 0 for both FCSD and FCVD.

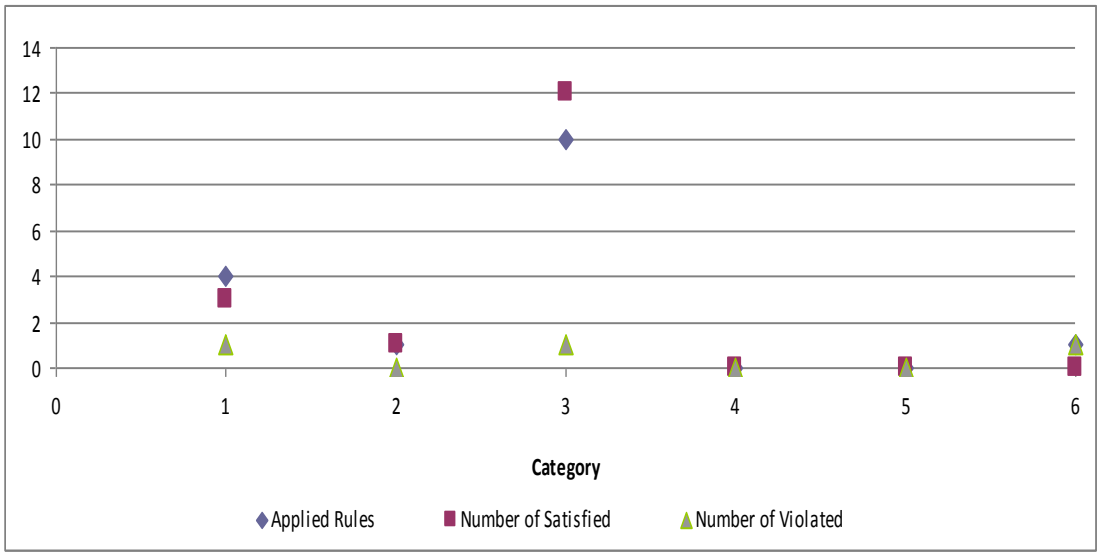

Figure 5.9 Applicable rule for function *one*

Figure 5.9 shows the Applicable Rules in the *one* function. In function *one*, Category 3 has the most number of Applicable Rules, which makes Category 3 have a

significant effect on the final FAC value of the function. The full Function Calculation Table of the *one* function is shown in Appendix F.

5.5.1.4 *main* Function

The function *main* is the function that is the entry point of the program calls all other functions in the program. With 27 Clauses, as shown in Appendix G, function *main* has the largest number of Clauses. 31 out of 37 rules have been identified as contributing to the Robustness Degree of the function *main*. These rules have been applied 63 times; 50 times they have been satisfied and 12 times they have been violated, which mean the FSAC = (50/63) % = **79.37%.**

For Categories 4 and 5, function *main* has 100% Function Category Satisfaction Degree (FCSD). Since the rules only apply once in these categories and they have all been satisfied, the number of Applicable Rules is equal to the number of times a rule is satisfied. Figure 5.10 shows the Applicable Rules in function *main*. It shows that the Category 2 has the most number of Applicable Rules in function *main*, which makes it the most effective category in the final FAC of function *main*.

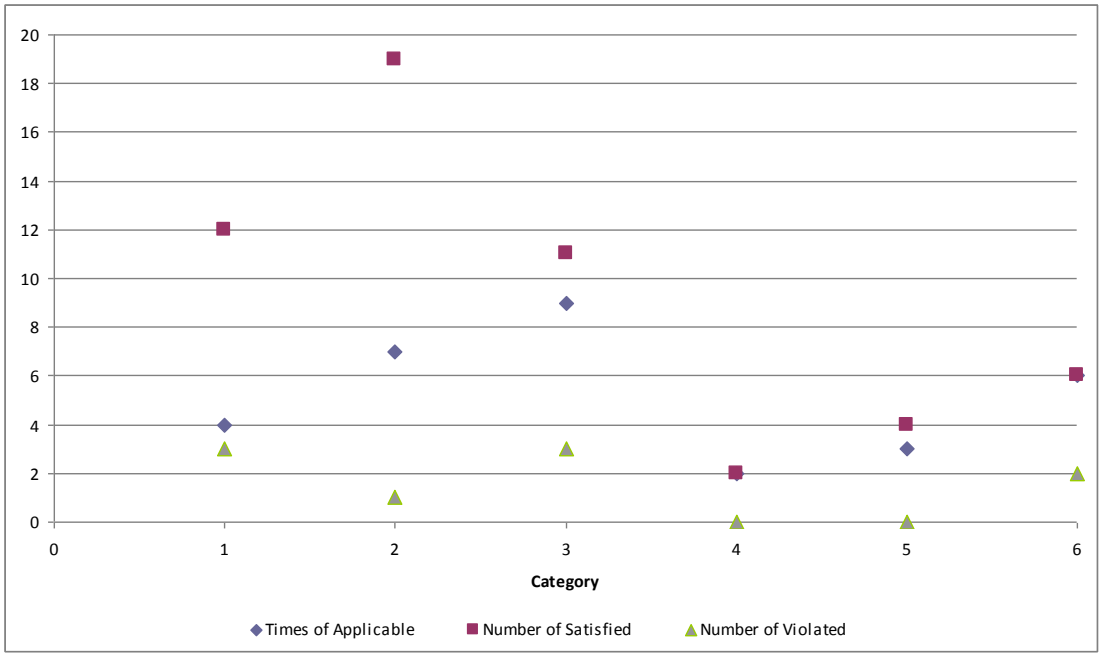

Figure 5.10 Applicable rule for function *main*

Function *main* has a high degree in Category 2 with 95%, where Categories 1, 3, and 6 score 80%, 78.57%, and 75%, respectively. The full Function Calculation Table of function *main* is shown in Appendix G.

5.5.2 Program Category Degree

Accumulative Category (AC) shows the function and the program behaviour through the Language Features Categories. It also gives the Robustness Degree of each function in all categories for the Function Accumulative Category (FAC). Figure 5.11 shows the FSAC for each function of the program, and the Whole Program Degree (WPD).

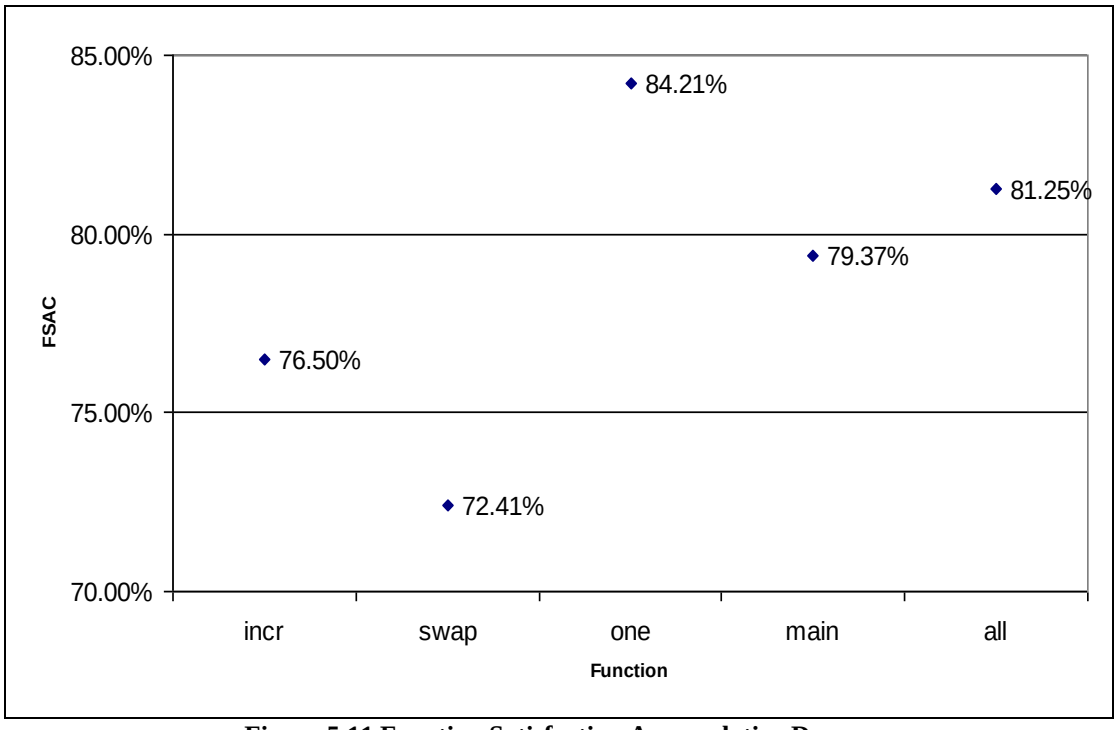

Figure 5.11 Function Satisfaction Accumolative Degree

Figure 5.11, shows that the *swap* function has the lowest Robustness Degree with FSAC = 72.41%, and it is the function that least satisfies the Language Features. The *one* function has the highest value with FSAC = 84.21%. As noticed, all functions' FSAC values are close to each other. This reflects on the Whole Program

Satisfied Degree (WPSD) value, which is equal to 81.25% and is almost in the middle between the highest and lowest Robustness Degrees scored by each function.

Program Category Degree (PCD) is the part of Robustness Grid that shows the Robustness measurement for all functions in the program. In Figure 5.12, the graph compares the Program Satisfied Category Degree (PSCD) with the Program Violated Category Degree (PVCD). It shows that all rules in Categories 4 and 5 are satisfied.

In Categories 1, 2, and 3, the PSCD has a larger Robustness Degrees over the PVCD for the same categories. However, in Category 6, the PVCD has a slightly larger value over the PSCD, since there are more rules that have been violated than satisfied.

Since all functions have a good Robustness Degree, the whole program has 81.25% as a Robustness Degree value.

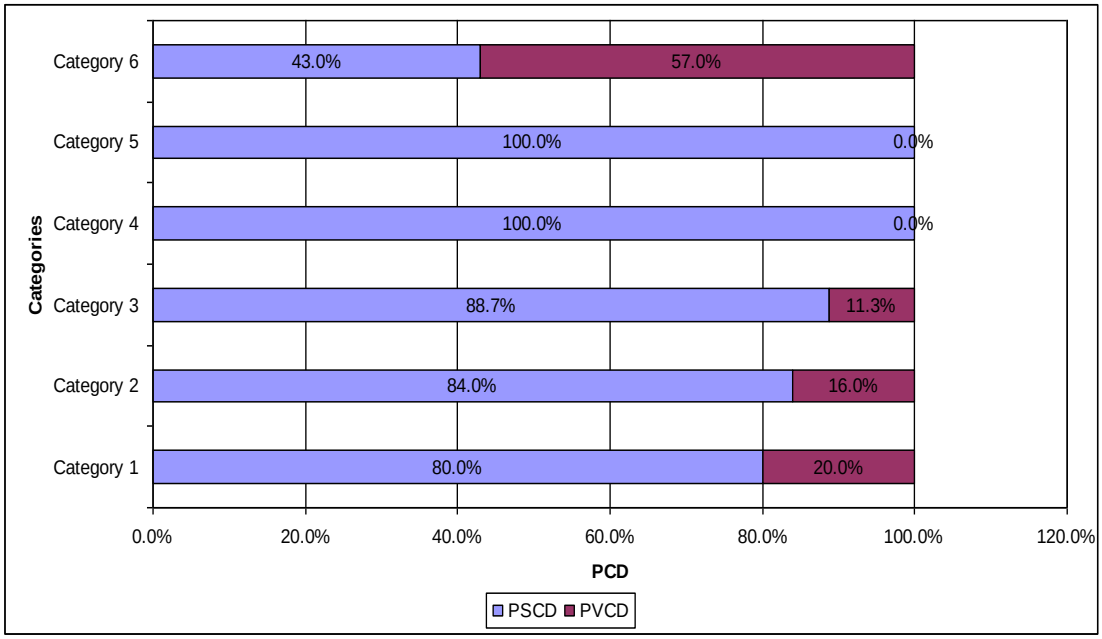

Figure 5.12 Comparison between PSCD and PVCD

5.5.3 Category Calculations

In Category Calculations, the Language Features characteristics: Category ∑ Slice Size, Category Frequency, and Rule Category Weight are used to identify the

Language Feature importance and the effect in the program. Figure 5.13 shows the Rule Category Weight that has been used to measure the SwapoAdd.c program and its functions.

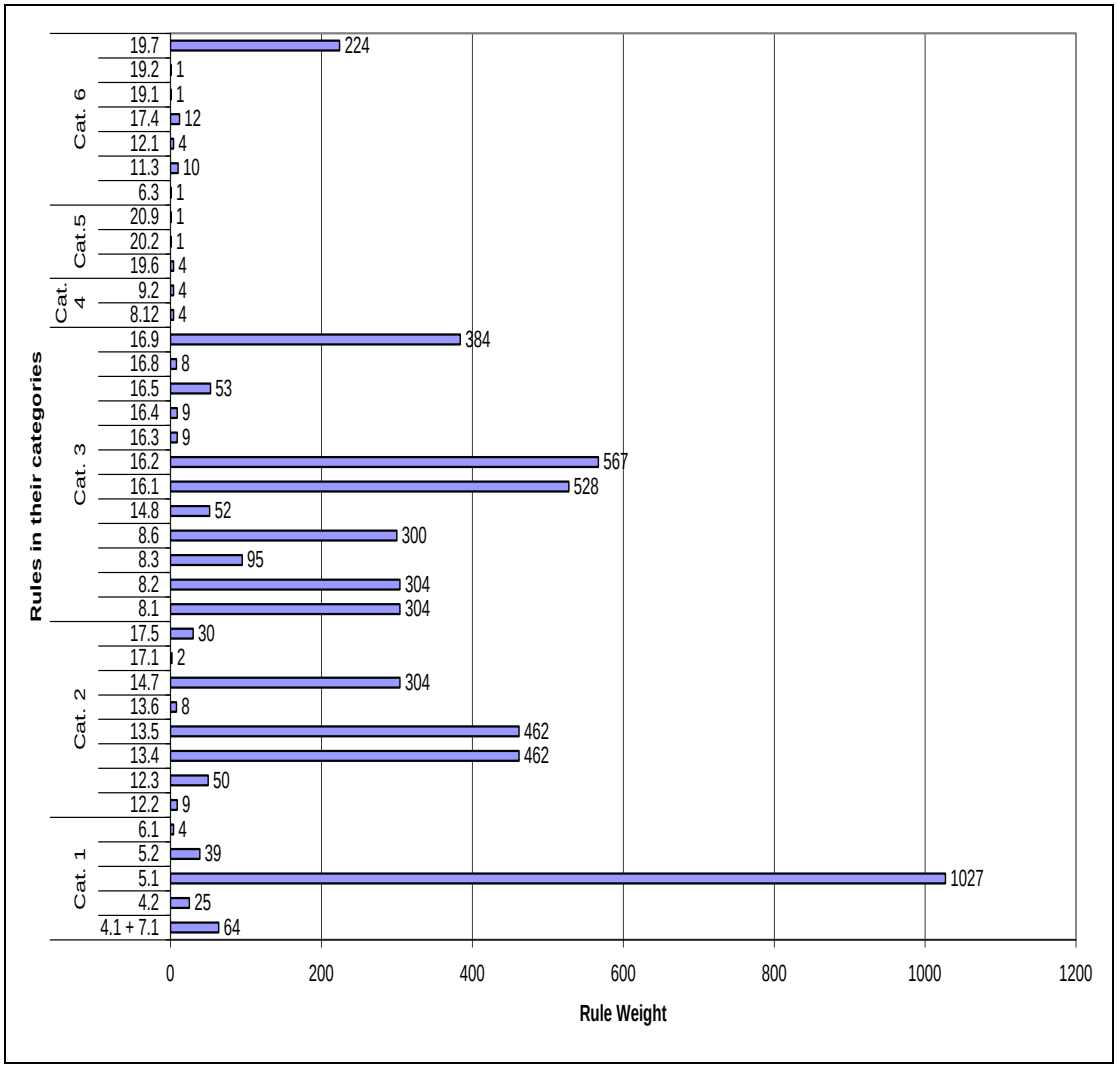

Figure 5.13 Rule Category Weight

It is noticeable that there are several rules that have small Weight values. Therefore, the change in these rules satisfaction status will not significantly affect the Robustness Degree. On the other hand, the rules with high Rule Category Weight values are the ones that affect the overall Robustness Degree. For example, Rule 5.1 will significantly affect the Robustness Degree if it is changed from being satisfied all the time to become violated. In such a case, the Robustness Degree will be

dropped to 71.09%. The Category Calculations give an indication as to which parts of the codes should be maintained first to raise the Robustness.

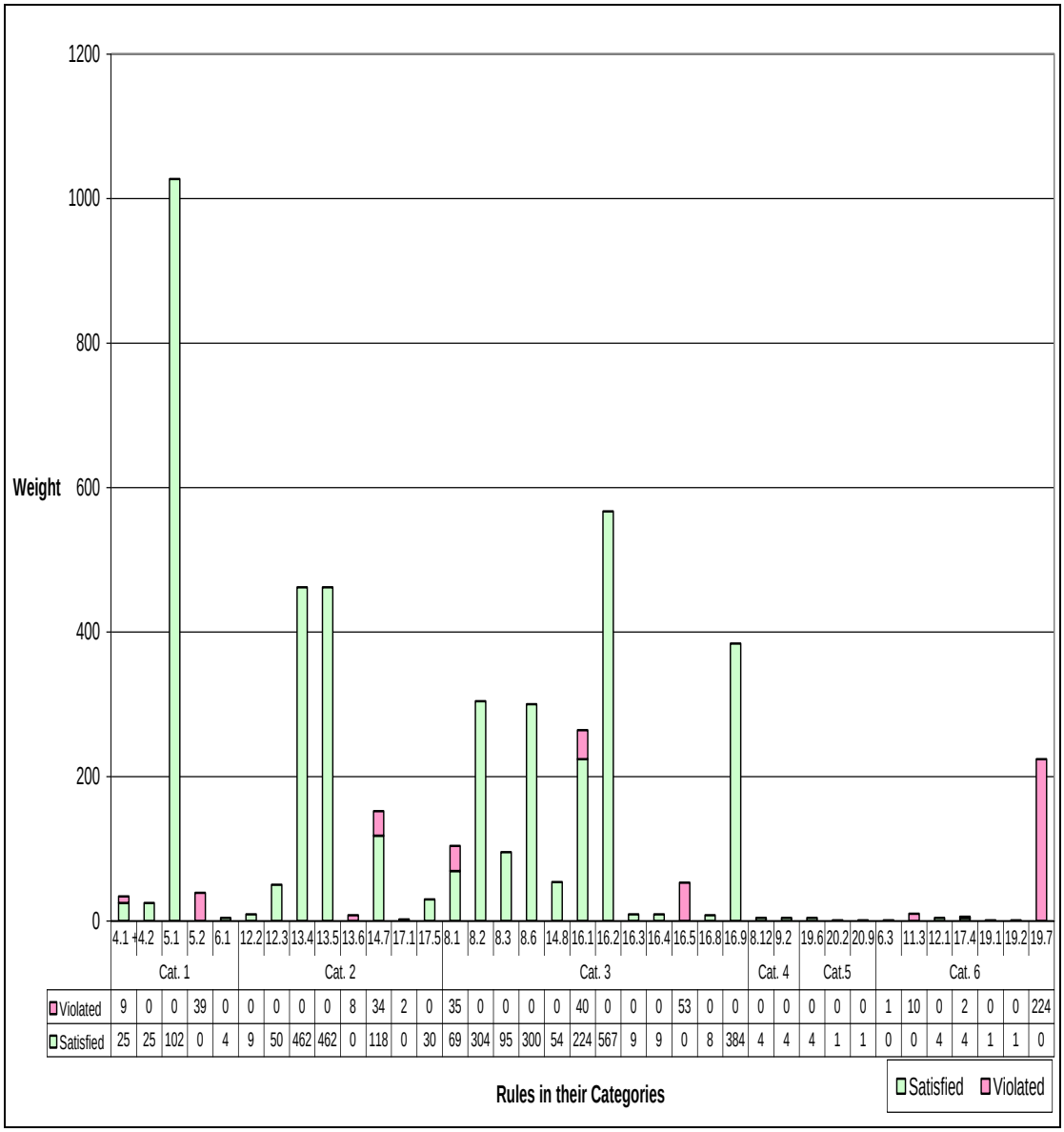

Figure 5.14 Satisfied/ Violated rules Weights

In Figure 5.14, the rules Satisfied and Violated Weight are shown. This Figure can help the maintainer to determine which rules should have the highest priority in the maintenance process. In the SwapoAdd.c program, Rule 19.7 (Category 6) comes first, since it has the largest Violated Weight, and then Rules 16.5, 16.1, and 8.1.

5.6 Results Analysis

In this section, an example of a function in the SwapoAdd.c program is shown to clarify how calculations are made. The critical issues will be pointed out, and the SwapoAdd.c program results will be explained.

5.6.1 Function *incr* as an example

In this section, the Robustness Grid will be built, step by step, for the *incr* function. The *incr* function has been chosen because it is the function with the smallest number of Clauses which make it an easy function to be used to explain the Robustness Grid construction steps.

Function *incr* has 8 Clauses, the Clauses and their numbers are as follows:

```
void incr(int *num, int i);[6]
incr[18](&sum[19], i[20]); }
void incr[39](int *num[40], int i[41]) {
*num = *num + i;}[42]
```

Each one of the Clauses and its number were used as the Slicing Criteria of the Clause Slice. The following Table 5.1 shows each one of these Clauses with its slice, where the Clauses and its slices are underlined.

In Table 5.1, Clause 6 is an un-sliceable Clause, so its slice is the clause itself. In the Clause Table for the function *incr*, the Clauses are measured against the MISRA C2 rules, and weighted using their Slice Sizes and Clause Frequency. Table 5.2 shows the Clause Table for Function *incr*.

The Clause	The Slice
void *incr*(int *num, int i);[6]	void *incr*(int *num, int i);[6]

incr[18](&sum[19], i[20]);}	*incr*[18](&sum[19], i[20]); }
	printf("sum = %d\n", sum[23]);
	void *incr*[39](int *num[40], int i[41])
	{
	*num = *num + i;}[42]

incr[18](&sum[19], i[20]); }	*incr*[18](&sum[19], i[20]); }
	printf("sum = %d\n", sum[23]);
	void *incr*[39](int *num[40], int i[41])
	{
	*num = *num + i;}[42]

incr[18](&sum[19], i[20]); }	*incr*[18](&sum[19], i[20]); }
	printf("sum = %d\n", sum[23]);
	void *incr*[39](int *num[40], int i[41])
	{
	*num = *num + i;}[42]

void *incr*[39](int *num[40], int i[41])
{
	printf("sum = %d\n", sum[23]);
	void *incr*[39](int *num[40], int i[41])
	{
	*num = *num + i;}[42]

void *incr*[39](int *num[40], int i[41])
{
	printf("sum = %d\n", sum[23]);
	void *incr*[39](int *num[40], int i[41])
	{
	*num = *num + i;}[42]

void *incr*[39](int *num[40], int i[41])
{
	printf("sum = %d\n", sum[23]);
	void *incr*[39](int *num[40], int i[41])
	{
	*num = *num + i;}[42]

*num = *num + i;}[42]	printf("sum = %d\n", sum[23]); *num = *num + i;}[42]

Table 5.1 *incr* Clauses slices

The Clauses Weight range in the *incr* function is between 16 and 48, except for Clause 6, which has the lowest weight value equal to 1 and that because it is an un-sliceable Clause.

Clause Number	Slice Size	Clause Frequency	Clause Weight	Function Name	Applicable Rules	
					Satisfied	Violated
6*	1	1	1	*incr*	16.3, 16.4, 16.1	19.7
18	8	6	48	*incr*	16.2, 16.9	13.6
19	4	5	20	*incr*	0	0
20	4	4	16	*incr*	0	0
39	5	7	35	*incr*	8.1, 8.2, 8.6	14.7
40	3	10	30	*incr*	5.1, 8.3	0
41	3	10	30	*incr*	5.1, 8.3	0
42	2	12	24	*incr*	12.2	17.1, 17.4
* Un-sliceable Clause						

Table 5.2 Clause Table for *incr*

The Data Table of the *incr* function is constructed using all the functions' Clause Tables, since the Data Table is related to the MISRA C rules which have been applicable in other program functions and their values are affected by them. Table 5.3 shows the Data Table of the Rules that have been applicable in the *incr* function. The Data Table of all the rules applicable in the program is given in Appendix D.

Rule Number	Number of Satisfied	∑Satisfied Slices	Number of Violated	∑Violated Slices
5.1	13	79	0	0
8.1	3	23	1	53
8.2	4	76	0	0
8.3	5	19	0	0
8.6	4	75	0	0
12.2	3	3	0	0
14.7	2	59	2	17
16.1	4	56	4	10
16.2	9	63	0	0
16.3	3	3	0	0
16.4	3	3	0	0
16.9	8	48	0	0
17.1	0	0	1	2
17.4	2	2	1	2
19.7	0	0	4	56

Table 5.3 Data Table of Applicable Rules in the *incr* Function

The *incr* Function Calculations, Table 5.4, use the Data Table, which provides the Satisfied and Violated Frequency and the Slice Size. The rest of the calculations are

derived from the Slice Size and the Clause Frequency. However, the Function Category Degree (FCD) is calculated using the Applied Rules column.

The Category Calculations depend on the Applicable Rules in the program and it can measure the Applicable Rules in a function. However, the Category Calculations do not measure the number of times the rules have been applicable in the function. This kind of measurement is made in the Function Calculations.

The overall FAC value of the function *incr* is 76.47% of applied rules have been satisfied and 23.53% have been violated.

In the *incr* Function, Rules 5.1 and 8.3 have the largest values in the Applied Rules, Satisfied, and ∑Satisfied Slice Sizes in Table 5.4 columns. On the other hand, the function has some rules with low values such as rules 16.3, 16.4, and 17.1. Rule 14.7 has the largest value in ∑Violated Slice Sizes and Violated Weight columns for *incr* function.

Categories	Rule Number	Incr								FCD %	
		Applied Rules	∑Satisfied Slice Sizes	Satisfied Weight	∑Violated Slice Sizes	Violated Weight	Function Frequency	Rule ∑ Function Slice Size	Rule Function Weight	FCSD %	FCVD %
Category 1	4.1 + 7.1	0	0	0	0	0	0	0	0	2/2 = 100%	0/2=0%
	4.2	0	0	0	0	0	0	0	0		
	5.1	+2	6	12	0	0	2	6	12		
	5.2	0	0	0	0	0	0	0	0		
	6.1	0	0	0	0	0	0	0	0		
AC	5	1	6	12	0	0	2	6	12	2/2 = 100%	0/2=0%
Category 2	12.2	+1	1	1	0	0	1	1	1	1/3 = 33.3%	2/3 = 66.7
	12.3	0	0	0	0	0	0	0	0		
	13.4	0	0	0	0	0	0	0	0		
	13.5	0	0	0	0	0	0	0	0		
	13.6	0	0	0	0	0	0	0	0		
	14.7	-1	0	0	5	5	1	5	5		
	17.1	-1	0	0	2	2	1	2	2		
	17.5	0	0	0	0	0	0	0	0		
AC 1-2	13	4	8	14	7	7	5	14	20	3/5 = 60%	3/5 = 60%
Category 3	8.1	+1	5	5	0	0	5	5	5	10/10 = 100%	0/10 = 0%
	8.2	+1	5	5	0	0	5	5	5		
	8.3	+2	6	12	0	0	6	6	12		
	8.6	+1	5	5	0	0	5	5	5		
	14.8	0	0	0	0	0	0	0	0		
	16.1	+1	1	1	0	0	1	1	1		
	16.2	+1	8	8	0	0	8	8	8		
	16.3	+1	1	1	0	0	1	1	1		
	16.4	+1	1	1	0	0	1	1	1		
	16.5	0	0	0	0	0	0	0	0		
	16.8	0	0	0	0	0	0	0	0		
	16.9	+1	8	8	0	0	8	8	8		
AC 1-3	25	13	40	60	7	7	45	54	66	13/15 = 86.7%	2/15 = 13.3%
Category 4	8.12	0	0	0	0	0	0	0	0	0	0
	9.2	0	0	0	0	0	0	0	0		
AC 1-4	27	13	40	60	7	7	45	54	66	13/15 = 86.7%	2/15 = 13.3%
Category 5	19.6	0	0	0	0	0	0	0	0	0	0
	20.2	0	0	0	0	0	0	0	0		
	20.9	0	0	0	0	0	0	0	0		
AC 1-5	30	13	40	60	7	7	45	54	66	13/15 = 86.7%	2/15 = 13.3%
Category 6	6.3	0	0	0	0	0	0	0	0	0/2 = 0%	2/2 = 100%
	11.3	0	0	0	0	0	0	0	0		
	12.1	0	0	0	0	0	0	0	0		
	17.4	-1	0	0	2	2	1	2	2		
	19.1	0	0	0	0	0	0	0	0		
	19.2	0	0	0	0	0	0	0	0		
	19.7	-1	0	0	1	1	1	1	1		
FAC	37	15	40	60	10	10	47	57	69	13/17 = 76.47%	4/17 = 23.53%

Table 5.4 *incr* Function Calculations

Table 5.5 shows the rules applicable in the *incr* function and their overall Category Calculations. Table 5.5 also shows that the *incr* Function has applied the most important applicable rule in the program, which has the biggest Weight Value among all other rules; Rule 5.1.

For the Function Calculations, Rule 5.1 has been applied and satisfied twice in the function. Moreover, almost half of the *incr* function applied rules are in mid-level weight values compare with all other rules, which lead to the conclusion that the *incr* function has a medium level of importance.

Rule Number	CATEGORY CALCULATIONS								
	Category Satisfied Frequency	Category Satisfied ∑Slice Sizes	Category Satisfied Weight	Category Violated Frequency	Category Violated Slice Sizes	Category Violated Weight	Category Frequency	Category ∑Slice Sizes	Rule Category Weight
5.1	13	79	1027	0	0	0	13	79	1027
12.2	3	3	9	0	0	0	3	3	9
14.7	2	59	118	2	17	34	4	76	304
17.1	0	0	0	1	2	2	1	2	2
8.1	3	23	69	1	53	35	4	76	304
8.2	4	76	304	0	0	0	4	76	304
8.3	5	19	95	0	0	0	5	19	95
8.6	4	75	300	0	0	0	4	75	300
16.1	4	56	224	4	10	40	8	66	528
16.2	9	63	567	0	0	0	9	63	567
16.3	3	3	9	0	0	0	3	3	9
16.4	3	3	9	0	0	0	3	3	9
16.9	8	48	384	0	0	0	8	48	384
17.4	2	2	4	1	2	2	3	4	12
19.7	0	0	0	4	56	224	4	56	224
SUM	73	509	3119	13	138	337	76	649	4078

Table 5.5 *incr* function Category Calculations

5.6.2 SwapoAdd.c Functions' behaviour

Robustness Grid presents the different functions evaluations. Table 5.6 shows a comparison between the SwapoAdd.c functions.

Function Factors \ Function Name	*incr*	*swap*	*one*	*main*
Number of Clauses	8	16	9	27
Applied Rules	15	16	16	31
Total Rule Frequency	17	29	19	63
FAC	76.5%	72.41%	84.21%	79.37%
Total Rules Function Weight using Function Calculations	69	414	88	2040
Function Weight using Category Calculations	4078	4193	4191	5202

Table 5.6 Comparison between SwapoAdd.c functions

Furthermore, it shows that the function *main* is the most important function in the program, since it has the biggest values mainly in the Function Weight.

Furthermore, the comparison shows that the Number for Clauses of a function has no relation with the Number Applied Rules for the same function. For example, the functions *one* and *swap* have 16 and 9 Clauses, respectively. However, they are equal in the number of Applied Rules. In addition, the number of Applied Rules has a minor effect on the Rule Function Weight. For the same functions, *swap* and *one*, the Rule Function Weight is significantly different although they have the same number of Applied Rules. On the other hand, Slice Size and Total Rule Frequency for a function are both the direct factors that produce the total Rules Function Weight, where the Function in the Category Calculation uses them for all Functions. It is for this reason which explains the difference between Function Weight in the Rule

Function Weight using Function Calculations or using the Category Calculations. The *main* function scores the highest weight, where as the *incr* function scores the lowest.

As a conclusion, it is not how many rules have been applied in the function, but indeed it is which rules have been applied on the function that indicates the function effect and importance.

5.7 Summary

This chapter has presented the Robustness Grid measurement of the SwapoAdd.c program, where the program has **81.25%** as the Robustness Degree scored in Program Satisfied Category Degree (PSCD), and the Whole Program Weight (WPW) is **5366**.

The Clause Table measures each clause in the program. It shows that Clause number 9 has the highest Weight value. The un-sliceable clauses have scored the lowest Weight and this is as expected since they have no slice but themselves, and they have not been in any other slice.

The Data Table measures the satisfaction status, and the number of times a rule has been applicable in the program. Rule number 5.1 has the highest number of times it has being applicable and satisfied. It also has the highest Satisfied Slice Size. On the other hand, there are some rules that measure only un-sliceable one clause. They scored low numbers of satisfaction times and Slice Size. Rule 5.1 is the Applicable Rule that has been satisfied the most, where 16.1 and 19.7 are the most violated rules.

The Robustness Grid is built step by step, where the Clauses are measured first, then the rules, then the Functions' Calculations is constructed. These steps only measure individual pieces of the program. Therefore, to produce the full Robustness Grid, these small measurements are accumulated by other Calculations: AC, FAC,

PCD, and Category Calculations. These calculations measure the program as one piece and from different points of view.

Table 5.7 shows a Managerial-View of the Robustness Grid, where only a summary of the main Robustness Degrees are presented. The Managerial-View Table is very useful to help the developer where it shows the function with its weight and Robustness Degrees which indicates the functions priority to be maintained to get a higher Robustness Degree. The function with high weight and lowest Robustness Degree is the Function that must be maintained first because it will have a significant effect on the rest of the program. The developer has the chose to either give the priority to the Function Rules Weight or to the Category Weight to put the maintenance plan. In Table 5.7 the function *swap* has been chosen to be maintained since it has the lowest Robustness Degree, with quite high Function Weight value. Category 6 has a low Robustness Degree but a low Category Weight Value, which has a lower effect than the *swap* function.

Rule Categories	*incr* FCSD	*swap* FCSD	*one* FCSD	*main* FCSD	PCD PCSD	Category Weight
Category 1	2/2 = 100%	7/9 = 77.78%	3/4 = 75%	12/15 = 80%	24/30 = 80%	1154
Category 2	1/3 = 33.3%	0/1 = 0%	1/1 = 100%	19/20 = 95%	21/25 = 84%	1332
Category 3	10/10 = 100%	14/16 = 87.5%	12/13 = 92.3%	11/14 = 78.57%	47/53 = 88.68%	2613
Category 4	N/A	N/A	N/A	2/2 = 100%	2/2 = 100%	8
Category 5	N/A	N/A	N/A	4/4 = 100%	4/4 = 100%	6
Category 6	0/2 = 0%	0/3 = 0%	0/1 = 0%	6/8 = 75%	6/14 = 43%	253
FAC	13/17 = 76.47%	21/29 = 72.41%	16/19 = 84.21%	50/63 = 79.37%	104/128 = 81.25%	WPW = 5366
Function Rules Weight / All Rules Weight	4078/5366 = 76%	4193/5366 = 78.14%	4191/5366 = 78.10%	5202/5366 = 96.94%		

Table 5.7 Managerial-View for SwapoAdd.c Robustness Grid

Chapter Six

Evaluation

6.1 Introduction

The Robustness Grid uses static analysis techniques to measure the Robustness Degree for C programs using MISRA C2 language rules. In this section, the Robustness Grid will be evaluated and compared with other techniques using the *SwapoAdd.c* measurement results.

6.2 Evaluation of the Robustness Grid

The Robustness Grid, as a program robustness measurement technique, succeeds by introducing the defect and dangerous code clauses that may cause a problem during program execution. The Robustness Grid consists of two tables: Clause Table and Data Table. The Clause Table highlights each clause with its characteristics such as: the clause effect, importance, and the rules that were violated or satisfied.

The Clause Table identifies the most effective clause by addressing the size of the Clause Slice.

In the Clause Table, which is one of Robustness Grid components, the Clauses of the program code are measured with their characteristics: the Slice Size, Clause Frequency, and Clause Weight. The clause information will make it easier to identify the most effective clause, and also the most important clause with the most effect in the program. The Clause Table also shows the MISRA C2 rules that were satisfied or violated by each clause of the program, which make it quite simple to see how each clause needs to be modified to make it more robust. Clause Slice is used to weight each code clause and MISRA C2 rule individually. It gives the developers and maintainers an indication of what they need to look at to increase the program robustness.

Furthermore, the Data Table, another part of Robustness Grid components, shows the measurement of each MISRA C2 rule. Each rule is measured depending on the number of times of satisfaction and violation, and the rules are weighted depending on the clause that applied the rules, where the rule that measures many clauses or important ones will be an important rule. The Data Table shows the rule importance for the program robustness and shows the rules with high effect on the Robustness Degree of the program, which make it easy to identify the defective rules that need to be maintained.

The Clauses are grouped together in the Robustness Grid to show the rules applied, the clauses' importance, and Robustness Degree of each function in the program presented as percentage. The function presentation, in the Function Calculation Table, shows the function with the most/least Robustness Degree, and the functions with large weights and the one with the most effect on the whole program Robustness Degree.

The Applied Rules make the developer check whether all the code lines were checked using the rules that the developer really wants, and see whether the code satisfied the rules or violated them. Because all rules categorised and rules in a category share the same theme. The category which highly violated shows the weak areas of programming. For example if Category 3 has the least satisfied Robustness Degree, then the developer should take care of the function's structure programming.

The Robustness Grid only uses 100 out of the 142 MISRA C2 rules, which is only around 70% of all rules. In addition, Robustness Grid interprets the MISRA C2 rules in different way that is used in other techniques, for example; in the Robustness Grid the library functions such as *printf* is considered as any other function. In such as case, the *printf*, and *scanf* functions are not robust. Still in the Robustness Grid, all C code standard libraries are entirely robust.

The Robustness Grid cannot be used to compare the Robustness Degrees of two functions or two programs. Each program applies different rules and has different code, which leads to incomparable slices and weights.

The Robustness Grid ignores the compiler warnings. The compiler warnings may be caused because of the violation of other MISRA C2 rules that are not included in the Robustness Grid measurement. However, the compiler warnings may be different for different compilers. Initially, the Robustness Grid only uses the gcc compiler, and that makes it more necessary to find a way to measure these warnings and identify the list of active and inactive warnings for the program for gcc or any other compiler used.

The robustness measurement results in the Robustness Grid give all levels of details, starting with the clauses or rules, leading to an abstract results report called Managerial-View. This report can give different types of results' presentation regarding the scope of focus; Functions, Categories, or Rules point of view.

6.3 SwapoAdd.c Evaluation

The Robustness Grid Managerial-View, Table 5.7 as an example, shows different numbers and scales that have different meanings. The Managerial-View shows, for each function in SwapoAdd.c, the number of rules that were applied and satisfied, and the percentage relations between them.

In the Managerial-View, the percentages show that the most important function in the *SwapoAdd.c* is the function *main*, as it has the largest weight and its slices contain 96.94% of all program clauses. Then, functions *swap* and *one* came in the second and third positions with very close percentages; 78.14% and 78.10%, respectively. Even though function *one* has 5 clauses less than *swap*, the importance of both function is almost similar. The reason is that the Clauses' Slice Sizes of function *one* is larger than the *swap* function clauses, and also function *one* has applied the same number of rules as the function *swap*, with slightly different types of rules, but almost have the same importance.

Even though *incr* is the least important function in the program, it is still important since its slices contain 76.47% of the program clauses.

Not all categories have rules applied in all functions. Categories 4 and 5 have only rules applied on function *main* since the function *main* is the only function that has code that is measured by all categories.

The categories of the Robustness Grid are shown in the Managerial-View with a percentage for the satisfaction degree of each function in each one of them. Furthermore, the Managerial-View shows the weight of each category. The Function Weight and the Category Weight is the scale used to rank the importance of each category in the program. Category 3 is the most important category in the program since it applied the largest number of rules and has the largest weight. Category 3

contains the rules that consider the function's structure, and the functions are taking a big share of the program, so the Robustness Grid is a quite reflective scale.

Regarding to the Managerial-View, the Category 2 is more important than Category 1; even though it has a smaller number of Applied Rules. This is because Category 2 has applied more important rules and has more effect in the program than Category 1. In SwapoAdd.c, Category 2 has the rules that consider the control statements, which has effect on the program more than the type definition and arithmetic statements that measured by Category 1. Categories 4 and 5 are only applied in the function main 2 and 4 times respectively, which lead to make them the least effective and importance in the Robustness Grid Calculations.

6.4 Robustness Measurement using other techniques

The *SwapoAdd.c* Robustness Degree was measured by two different techniques that use MISRA C2 rules: LDRA TBmisra and FlexeLint in addition to the Robustness Grid. The Robustness measurement results by LDRA TBmisra and FlexeLint were different from the Robustness Grid. However, Klocwork Truepath is a tool that uses MISRA C2 as well but it was not accessible to measure the SwapoAdd.c. Klocwork Truepath is compared with other tools: Robustness Grid, LDRA TBmisra, and FlexeLint regarding to the measurement principles and procedure.

Moreover, Gallagher and Fulton (Gallagher and Fulton 1999) tried to estimate the program robustness using a fault injection technique employing program slicing. Their measurement was also compared to the Robustness Grid.

In this section, the different techniques and their results will be presented and used to evaluate the Robustness Grid.

6.4.1 LDRA TBmisra

In the LDRA evaluation, TBmisra (LDRA 2012) was used to evaluate the program against MISRA C2 rules. TBmisra uses all the 147 MISRA C2 rules to measure the Robustness Degree. Even TBmisra shows which rules have been violated; it does not show which of the 147 rules were applied and which were not. Furthermore, the results from TBmisra did not show the effect of these violations on the rest of the program. Therefore, all rules in the LDRA evaluation have the same weight, and the results do not identify the rule that has a major or minor bad effect on program robustness.

In the LDRA TBmisra evaluation results of the *SwapoAdd.c* program; see Appendix M, 9 MISRA C2 rules were violated at least once, which is less than what was violated in Robustness Grid, where 12 rules were violated. However in the LDRA evaluation, the number of times the rules were violated is 33 times, which is larger than what is in the Robustness Grid with 24 times.

In the Robustness Grid Clause Table and LDRA TBvision, the developer can access to the code and see for each clause or line in TBvision which rules were violated. The satisfied rules can also be seen in the Clause Table.

A high level of MISRA C2 evaluation results are displayed differently by both of LDRA TBvisoin and the Robustness Grid. LDRA TBvisoin shows the rules that were violated and how many times they were violated. In the Managerial-View, see page 111, which is a summary of the Robustness Grid, the number of violated and satisfied rules are shown with the percentage of the Robustness Degree of each Function and Category in the program.

The LDRA TBmisra evaluation results show that Rule 6.3 is the most violated rule, with 17 times, which means that the most violated rule in the *SwapoAdd.c* program was the advisory rule number 6.3. On the other hand, this was the only rule that the

Robustness Grid and LDRA TBmisra agreed that it was violated, but it was violated only once according to the Robustness Grid. Since the LDRA evaluation has included all the MISRA C2 rules, unlike the Robustness Grid, which has Rule 21.1 that was violated but it is outside the scope of the Robustness Grid.

The LDRA TBmisra evaluation and the Robustness Grid have a disagreement on the violation of three rules, where they have been addressed as violated rules by LDRA, and satisfied by Robustness Grid. These rules are: 9.2, 20.9, and 4.2. Furthermore, the Robustness Grid has identified 11 rules that were violated where LDRA did not. This disagreement is understandable since the rules are written in plain English language, which leads to different interpretations and different results. However, this misunderstanding could be harmful, and the MISRA C2 rules could lose their value as standard rules, since the aim of standards is to identify one possible way to write and understand the code.

In *SwapoAdd.c*, the LDRA TBmisra evaluation results show that 8 Mandatory Rules and 25 Optional Rules were violated and no Checking Rules were reported.

The Static and Dynamic analyses are not fully covered by the Robustness Grid or MISRA C2, so the LDRA created their own set of rules (800 rules) that examine MISRA C2 rules in details in addition to Static and Dynamic analysis. In *SwapoAdd.c*, an evaluation report made by LDRA TBmisra indicates that the program has 22 Static violation, 3 Static Dataflows, and 8 Full Variable Cross Reference violations. These violations were not fully discovered by the MISRA C2 rules. The LDRA rules reported 2 Static Mandatory violations that were not caught by MISRA C2. These violations are about the use of pointer arithmetic. In the Robustness Grid, the static analysis is done by the MISRA C2 rules for the program code producing Rule Satisfaction Status, where the Dynamic analysis is made by the Clause Slicing technique producing the Clauses and Rule Slice Size and Weight.

LDRA TBmisra presents the results in a file where each line of code is followed by a line of the type of violation, where the violation line as follows:

```
(M, C, or O) <Analysis violation>: #LDRA rule (symbol of analysis
violation): <MISRA C2 violated rule> <error details>
```

One of the assumptions that the program should have, before being measured by the Robustness Grid, is that the program was programmed following the ISO 1990 C standards. The LDRA **Testbed** evaluated the *SwapoAdd.c* against ISO 1990 C language standards. The measurement returned some errors and warnings addressed in Appendix O. However, the gcc compiler compiled the program with only warnings. Since the compiler warnings are ignored, the *SwapoAdd.c* program was accepted to be measured by the Robustness Grid.

In addition to LDRA TBmisra and TBvision, LDRA introduced a compiler called **TenDRA**. TenDRA compiled the *SwapoAdd.c* program using ISO: C90 standards and reported 5 errors and 9 warnings, though it was compiled by gcc and reported only 5 warnings. ISO: C90 is used to produce MISRA C2 rules and the program is supposed to be written according to them.

6.4.2 FlexeLint

The FlexeLint measured the *SwapoAdd.c* program using MISRA C2 rules according to their interpretation. FlexeLint is similar to LDRA TBmisra, where both of them did not show which rules were applied or satisfied and only showed the ones that were violated. FlexeLint only applies static analysis on the assessment, since they applied only a static rule to evaluate the program robustness.

According to the FlexeLint measurement, shown in Appendix N, *SwapoAdd.c* violated 8 MISRA C2 rules a total of 44 times which is higher than LDRA and 33 times and Robustness Grid with 28. This is predictable because FlexeLint does not use any compiler and have any rule violated defined by gcc in the Robustness Grid

are also include. Only one of these 8 violated rules was an advisory rule: 6.3. The rest, the required rules, belong to a different set of MISRA C2 rules (1.2, 4.2, 8.4, 10.2, 12.13, 14.13, and 16.10). The 6.3 rule was violated 16 times, which make it the most violated rule in the FlexeLint analysis. Although it is also the most violated rule from LDRA TBmisra, the Robustness Grid reported that it was violated once.

Rule 1.2 was violated 11 times, which make it the most violated rule among the required rules. However, the Robustness Grid did not include this rule. Furthermore, the LDRA evaluation did not mention this rule as one of the violated rules. As mentioned before, it is not shown in the LDRA evaluation whether Rule 1.2 was used to evaluate *SwapoAdd.c* or not, because if it was applied then it must be satisfied.

The FlexeLint, LDRA TBmisra and the Robustness Grid agreed on two MISRA C2 rules: 6.3 and 4.2. However, they disagreed on the number of times 6.3 was violated and the satisfaction status of 4.2. In the Robustness Grid, rule 6.3 was violated once, where in FlexeLint and LDRA TBmisra it was violated 16 and 17 times respectively. Rule 4.2 was satisfied 5 times in the Robustness Grid and was violated once in FlexeLint and LDRA TBmisra.

Moreover, FlexeLint reported 2 rules: 10.2 and 12.13 that were violated in *SwapoAdd.c* but not included in the Robustness Grid. On the other hand, 12 rules were violated in the *SwapoAdd.c* according to the Robustness Grid but not included in the FlexeLint results.

Since the FlexeLint was using all 142 MISRA C2 rules and did not use a compiler, 4 rules were violated that are out of the Robustness Grid rules scope (see Robustness Language Features Conditions in Section 3.2.2.1). The FlexeLint measurement report showed the MISRA C2 violated rules' number and type, not as the results as in LDRA TBmisra report.

In the *SwapoAdd.c* results, only Syntax Errors, Warnings, Informational, and Elective Notes are shown. The FlexeLint also showed a *Wrap-up* which is a summary for each module (any .c file) in the program and it shows the errors in that module. In addition, FlexeLint shows a *Global Warp-up* that contains the main rules that were violated by the program, and the completion status; whether successful or failed and the number of messages produced for all modules. In *SwapoAdd.c*, the completion was successful and 57 messages were produced.

In the FlexeLint results, it reported that the violation of some rules that cause compiler time errors since it does not use a compiler in the assessment process. For the same reason, it can assess a part of the program code, the same as LDRA TBmisra, where as the Robustness Grid cannot. It also provides a summary at the end of the results report, but FlexeLint only gives a summary for each file of the program, not for each function as in LDRA TBvision or each Function and Rule as in the Robustness Grid. However, neither LDRA TBmisra nor FlexeLint produced a numerical measurement for the program Robustness Degree, nor presented the program Robustness Degree as a scale or a percentage as in the Robustness Grid. The FlexeLint results report presents the line of code and then the violated FlexeLint rules followed by each equivalent MISRA C2 rule and its type; whether it is required or advisory. The line is as follows:

```
The code line
Module name: code line number: column number: FlexeLint rule type and
number: error text [MISRA C2 rule number, required/advisory]
```

Here is an example:

```
void swap(int *a, int *b);
SwapoAdd.c:7:14: Note 970: Use of modifier or type 'int' outside of
a typedef [MISRA 2004 Rule 6.3, advisory]
```

The difference in the results for the SwapoAdd.c robustness measurement for the Robustness Grid and other tools is due to the different interpretation and application

of the MISRA C2 rules. Table 6.1 shows a comparison between the robustness measurement results of the Robustness Grid, LDRA TBmisra, and FlexeLint for the *SwapoAdd.c* robustness measurement.

In SwapoAdd.c measurement			
	Robustness Grid	*LDRA TBmisra*	*FlexeLint*
MISRA C2 rules used to evaluate the program	100 rules	All (142)	All (142)
MISRA C2 rules applied in the program	37	Not given	Not given
Number of rules Satisfied in the entire program	25	Not given	Not given
Number of rules violated at least once	12	9	8
Number of times the rules were violated	24	33	44
The most satisfied rule(s)	5.1, (13 times)	Not given	Not given
The most violated rule(s)	16.1, 19.7 (4 times)	6.3 (17 times)	6.3 (16 times)
Rules applied and outside Robustness Grid Scope	None	21.1	1.2, 8.4, 16.10, 14.3
Rules in the tool measurement but not in Robustness Grid	None	8.7, 20.8, 20.12, 5.7	10.2, 12.13
Rules violated in the tool measurement but not in Robustness Grid	None	9.2, 20.9, 4.2	10.2, 12.13

Table 6.1 Comparison between the Robustness Grid, LDRA TBmisra, and FlexeLint

Table 6.2 shows the rules that SwapoAdd.c has violated using the Robustness Grid, LDRA TBmisra, and FlexeLint. In the Robustness Grid, has applied some rules but they were not violated, and given (0) value. In Table 6.2, the rules that not included in the Robustness Grid are pointed as (N/I), and the not applied as (NA) and the satisfied with (0). The numbers mean the number of time a rule been violated. In LDRA and FlexeLint the applied or satisfied rules can not be distinguished, but they for sure were included, so the rules with such case have been pointed out as (0/NA).

MISRA C2 Rule	Robustness Grid	LDRA TBmisra	FlexeLint
1.2	N/I	0/NA	13
4.1+7.1	3	0/NA	NA
4.2	0	1	1
5.1	0	0/NA	0/NA
5.2	3	0/NA	0/NA
5.3	NA	0/NA	0/NA
5.7	NA	8	0/NA
6.3	1	17	16
8.1	1	0/NA	0/NA
8.4	N/I	0/NA	4
8.7	NA	1	0/NA
9.2	0	1	0/NA
10.2	NA	0/NA	2
11.3	2	0/NA	0/NA
12.13	NA	0/NA	1
13.6	1	0/NA	0/NA
14.3	N/I	0/NA	1
14.7	2	0/NA	0/NA
16.1	4	0/NA	0/NA
16.5	1	0/NA	0/NA
16.10	N/I	0/NA	6
17.1	1	0/NA	0/NA
17.4	1	0/NA	0/NA
19.7	4	0/NA	0/NA
20.8	NA	1	0/NA
20.9	0	1	0/NA
20.12	NA	1	0/NA
21.1	N/I	2	0/NA

Table 6.2 Number of MISRA rules that violated in SwapoAdd.c

6.4.3 Klocwork Truepath

As for the Robustness Grid, LDRA TBmisra, and FlexeLint, Klocwork Truepath uses its own interpretation of the MISRA C2 rules (Klocwork 2012). Klocwork has a managerial report, similar to the Robustness Grid, LDRA TBmisra, and FlexeLint where it shows a summary of the rules that were violated by the program.

There are 21 of the MISRA C2 rules that are not supported by Klocwork Truepath, 15 of them are not verified, and the rest are not supported even though they are verified.

	Robustness Grid	LDRA	FlexeLint	Klocwork Truepath
Standards used	MISRA C2	MISRA C2, ISO 1990, LDRA standards	MISRA C2, FlexeLint/PC-Lint	MISRA C2
Using Compiler before the measurement	Yes	Yes	No	Yes
Rule Weighing	Clause slice used to Weight the rules.	Mandatory Rules Optional Rules Checking Rules	Syntax errors Internal errors Fatal errors Warning messages Informational messages Elective notes	Not given
Management View	Yes	Yes	Yes	No
Map the Violated Rules with code line	Yes, using Clause Table.	Yes	Yes	Yes
Level of Faults	One level – Violated	Level 1: Advisory Level 2: Defect Level 3: Fault	Error Warning Informational Note	One level
Number of MISRA C2 supported	100	142 (all)	142 (all)	121

Table 6.3 Comparison between 4 robustness measurement techniques that use MISRA C2

16 MISRA C2 rules are not supported by The Robustness Grid or Klocwork Truepath. The Robustness Grid supports 5 rules that Klocwork Truepath does not support. On the other hand, Klocwork Truepath supports 26 rules that Robustness Grid does not support.

Table 6.3 shows a comparison between the different ways of the MISRA C2 use of the techniques and tools mentioned earlier.

6.5 Other case studies

6.5.1 Variance.c program

Gallagher and Fulton (Gallagher and Fulton 1999) used program slicing to estimate Software Robustness. They measured the Robustness of a C program by

determining the location of **high-impact** points and the exact nature of the error that could have a high impact.

In their work, they used the Forward Slicing technique to identify the high-impact point, where the most effective statement is the variable which has the biggest Forward Slice. Then, they applied a Decomposition Slice on that variable and injected an error in the last definition of it to identify the nature of the errors that could occur at this point.

The effect of the fault injection point was determined using the Forward Slicing techniques. Then, a random input was entered to the fault injection point, and the values of outputs determined how robust the program was. For each variable, which produces an output and is affected by the fault injection point, the Robustness is the percentage of invalid input that still produced an acceptable output.

The same program was measured by the Robustness Grid, where each clause was tested individually and its impact was measured using the Clause Slicing technique. A Clause Slice is used to determine the impact of each clause, where the clause with the largest slice is considered as the high-impact point in the program. Since Clause Slicing is used in the Robustness Grid, the clause may or may not have an effect on the output clauses. Furthermore, the Robustness Grid measures the robustness of a program depending on the MISRA C2 rules and their satisfaction through the program without looking as the output.

In the Variance.c program, see Appendix P, Gallagher and Fulton chose *avg* at statement 18 as the fault injection point to measure the robustness of *var1*, *var2*, *var3*, *var4*, and *var5*. Since Forward Slice of that statement did not include *var2*, the change of input of the *avg* value did not affect it, and the Robustness of *var2* was 1, which means 100%.

Since other variables; *var1*, *var3*, *var4*, and *var5*, are affected by *avg*, the Robustness was calculated by running the program with random values of *avg*. *var1*, *var3*, and *var4* have a robustness value = 0 because they always return an unacceptable output for the invalid input.

However, *var5* had a different value of robustness, where it succeeded to compute the correct value for 4.3% of values that were randomly perturbed.

In the Robustness Grid, the Robustness Degree for these variables (var1, var2, var3, var4, and var5) where measured differently. Each of these variables was sliced using Backward Clause Slicing (the union of all clauses that have effect on these variables) in their last use in the program. Then, the Robustness Degree for the slices of each one of these variable is given by:

Number of times the rules were satisfied divided by number of times the rules were applicable, presented as a percentage.

In the Robustness Grid, the Variance.c variables in general have high Robustness Degree. Furthermore, it pointed out the variable that has the priority to be maintained among the five variables (*var1*, *var2*, *var3*, *var4*, and *var5*) is *var3* because it has the smallest Robustness Degree. The variables; *var1*, *var2*, *var3*, *var4*, and *var5*, have a Robustness Degree of 90.91%, 91.17%, 87.88%, 90.63%, and 89.8% respectively. In addition, the *avg* has scored 80% as Robustness Degree, which make it the variable with the lowest Robustness Degree and it is the variable that should be maintained firstly. In the Robustness Grid the whole program scores 87.23%, while in Keith and Fulton technique has no overall robustness measurement for the program.

Gallagher and Fulton concluded that the Robustness needs a manual inspection for a specific variable within a scope of influence to compute the Robustness measurement. On the other hand, The Robustness Grid computes the Robustness Degree and gives a measurement to any variable in different scopes.

Gallagher and Fulton are more concerned about the input/output relation in the program; where the Robustness Grid is focused on the syntax code style of the program.

6.5.2 n_char.c program

The n_char.c (Drexel University 2012), see Appendix Q, was chosen to measure the accuracy of the Robustness Grid measurement. The n_char.c program has two functions: *n_char* and *main*. The program reads a string and prints its length.

As shown in Table 6.4, the overall program Robustness Degree is 77.78%. The functions' Robustness Degrees for functions *n_char* and *main* are 74.04% and 81.48%, respectively. The Functions Robustness Degrees are acceptable for a small and simple program such as n_char.c. Considering the Function Satisfaction Degree the Robustness Grid gives an advice that the function that need to be maintained is *n_char* because it is the function with the smallest Robustness Degree.

Rule Categories	n_char	main	PCD	Category Weight
	FCSD	FCSD	PCSD	
Category 1	83.33%	100%	90%	96
Category 2	66.67%	0%	57.14%	109
Category 3	75%	62.5%	70%	747
Category 4	N/A	100%	100%	13
Category 5	100%	100%	100%	170
Category 6	0%	80%	66.67%	37
FAC	74.04%	81.48%	77.78%	WPW = 1172

Table 6.4 n_char Robustness Grid, Managerial-View

However, the Robustness Degree in different categories for the individual functions varied significantly. The Robustness Grid shows that Category 3 is the most important category for this program because it has the largest weight and 70% Robustness Degree. Category 3 measures 63.74% of all programs clauses, which means that 63.74% of program clauses have 70% Robustness Degree.

Category 5, which is in the second place in terms of category importance, is 100% robust according to the Robustness Grid. Consequently, the functions and important categories have a high Robustness Degree, which is acceptable for such a small program with simple functionality. The Category Calculations indicates that the need to be maintained is category 2 because it is the category with the lowest Program Category Degree (PCD), and its weight makes the change effective.

6.5.3 *Robost.c* program

The *Robost.c,* Figure 6.1, is the smallest robust C program, according to the Robustness Grid measurement. *Robost.c* has 2 clauses, but it applied and satisfied 7 rules in 3 categories.

```
void main (void){[1]
return;}[2]
```

Figure 6.1 *Robost.c* program

Categories 1, 4 and 5 have no applied rules. Category 1 focuses on the variable characteristics, and because there are no variables or parameters, there is no need for rules to measure them. Category 4 measures the arrays, pointers and other data structures and *Robost.c* does not have these, so there are no applied rules in Category 4. There are no header files or pre-processor clauses so the Category 5 has no applicable rules in the program.

Table 6.5 shows the Managerial-View Robustness Grid report for *Robost.c*. Category 2 measures the control Clauses and *Robost.c* has no control Clauses. However, there is one rule that was applied; 14.7. This rule is for any function that should have one single point of exit, and because the program has a *return* Clause, it satisfied this rule. Other categories focus on function structure, and advisory rules which contain different points of views. Since the program is artificial, small, and ideal, the values are relatively small.

Rule Categories	*main*	*PCD*	Cat. Weight
	FCSD	PCSD	
Category 1	N/A	N/A	N/A
Category 2	1/1 = 100%	1/1 = 100%	1
Category 3	5/5 = 100%	5/5 = 100%	5
Category 4	N/A	N/A	N/A
Category 5	N/A	N/A	N/A
Category 6	1/1 = 100%	1/1 = 100%	1
FAC	7/7 = 100%	7/7 = 100%	WPW = 7

Table 6.5 *Robost.c* Robustness Grid Managerial-View

Robost.c shows that MISRA C2 covers even the smallest and basic program with a number of rules even more than the number of program clauses. MISRA C2 rules cover C programs regardless of their number of clauses. However, large programs are quite difficult to measure and take a considerably long time, in addition to the fact that Robustness Grid is produced manually. Also, in all programs, the most important clause is the main function call, since it calls all other functions and affects all of them.

6.6 Summary

The Robustness Grid, as a program robustness measurement technique, has some positives points and drawbacks. The contribution to research that have made in the Robustness Grid can be summarised as:

1- Using program analysis technique, clause slicing, to give a different level of importance for code and rules.

2- The numerical details of program Robustness Degrees are the main contribution that the Robustness Grid has introduced.

3- The Robustness Grid presentation is accessible and easy to understand for all level of developer teams. The program is analysed and the robustness degree is shown for each part of it.

However, there are still some weaknesses that need to be fixed in future work, such as:

1- The MISRA C2 rules misinterpretations

2- The number of rules that the Robustness Grid covers are only 100 out of 142 rules.

3- The Robustness Grid not fully automated.

The program measurement results show some agreement and disagreement between the tools that use MISRA C2 and the technique the use Slicing to measure the program robustness; even though they use the same standard.

LDRA TBmisra, FlexeLint, and Klocwork Truepath have introduced their own rules beside the MISRA C2 rules to avoid the misinterpretation of the MISRA C2 rules by the different programs evaluations. However, the interpretations by these three tools: are different. This was seen in SwapoAdd.c example, where the three techniques Robustness Grid, LDRA TBmisra, and FlexeLint had some differences in the rules that were violated, the number of times violated, and the code that caused the violation.

The different interpretation between different tools is dangerous and makes the standards less effective. However, LDRA and FlexeLint both introduced their own rules that interpret MISRA C2 in a certain way and make their measurement fixed in all the evaluations made. In the Robustness Grid, the MISRA C2 rules were interpreted in English without introducing new rules or setting fixed definitions and mainly depend on the MISRA C2 rules text and explanation for them which caused some differences between the Robustness Grid measurements and the measurements by other tools.

Chapter Seven

Conclusions and Future Work

7.1 Introduction

In this chapter, a summary of the research contributions will be shown. It also reviews the criteria for success defined in Chapter 1 as they related to the research.

The future directions for the research are suggested in this chapter, showing the possibilities for more contribution in this research area.

7.2 Thesis Summary

Robustness is defined as "*the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions*" (IEEE 1990). The Robustness Degree of a program is a value and can be measured in different ways using different techniques. Robustness is important in all programs, specially the safety critical ones. Therefore, it has become essential to measure the

Robustness Degree for programs and discover any weak points that the program has. There are many techniques, discussed in Chapter 2, that are used to satisfy or measure the Robustness Degree. Each one of those techniques has its advantages and drawbacks.

In this research, a group of the MISRA C2 language rules were selected with categories confirming to certain criteria (Section 3.2.2), to be the set of standards used to measure program robustness. The program satisfaction and violation status of the MISRA C2 rules are shown using numbers, called the Robustness Degree. The Robustness Degree was introduced in Section 3.4 as a "*scale of a program robustness features satisfaction, expressed as a percentage*" (Abdallah, Munro et al. 2011). The Robustness Degree of a program is presented in the Robustness Grid.

A Program Slicing technique is used to analyse the program syntax code to distinguish the important level of the language standards. Clause Slicing is a new slicing technique that was introduced in Section 3.3 to weight the importance of the MISRA C2 rules.

Clause Slice is a Static slicing technique that uses clause and clause number as the slicing criteria. A clause is a small piece of program code that has an effect in the program. It can be sliceable or un-sliceable, depending on what it contains.

MISRA C2 and Clause Slice were both used to formulate the Robustness Grid measurement technique. The measurement process in completing the Robustness Grid for functions was discussed through Section 3.4.

The Robustness Grid was extended in Section 3.5 to measure the Robustness Degree of the whole program. A Robustness Grid is a "*table that contains rules classified into categories, with respect to a program's function names and calculates robustness degree"* (Abdallah, Munro et al. 2011). It also shows the different levels of importance between the rules by weighting them using Clause Slice. The different

levels of rule satisfaction and weights are presented in numbers and percentages called the Robustness Degree, and that is one of the main contributions of this research.

The Robustness Grid contains the rules that were applied to measure a program and the state of the rules, whether satisfied or violated, for each function in the program. It also has percentage values that show the Robustness Degree for the program and each function in it in detail.

The Robustness Grid construction process starts with the Clause Table, which was shown in Section 3.4.2.1. The Clause Table contains the program clauses with their measurement information, such as Slice Size, Clause Frequency, and Applicable Rules. The Second Table in the Robustness Grid construction is the Data Table, which was addressed in Section 3.4.2.2.

The Data Table shows the MISRA C2 rules that were applied on the program. The rules are measured with respect to their Number of Satisfied and Violated, the $\sum$ Satisfied and Violated Slices.

The Clause Table was imported in the Robustness Grid to identify the rules that were used to measure every function in the program. The Function Calculations show the rules that were applied on each function and their degree of satisfaction. They also show the Function Satisfy Degree (FSD) and Program Satisfy Degree (PSD).

The Data Table was also imported in the Robustness Grid is used to measure the functions, and also used to produce the Category Calculations in the Robustness Grid, which measures the MISRA C2 rules.

The Category Calculations in Section 3.5.2 show the importance of each function, category, and rule in the program assessment. The Category Calculations are based on Clause Slicing.

The numbers and percentages in the Robustness Grid show the satisfaction state of each rule, and Robustness Degrees for functions, categories, and the whole program.

The Robustness Grid is not fully automated. However, some tools are used to execute some parts of the Robustness Grid process. The Clause Slice is run using the CSurf tool. The program clauses are measured manually against the MISRA C2 rules. The MS Excel is used to display the Clause Table, the Data Table, and the Robustness Grid and calculate the numbers and the percentages.

The validity of the Robustness Grid is explored through the application of different case studies that were discussed in details through Chapter 6. In this thesis, three case studies were discussed to validate and evaluate the Robustness Grid. The SwapoAdd.c program is the case study that was studied in detail through Chapters 3 and 5, and used to explain the Robustness Grid process in details. Other case studies were used to evaluate the Robustness Grid and addressed in Section 6.4.

The case studies show that the Robustness Grid can give a robustness measurement for C programs using the MISRA C2 rules. They also show that the Robustness Grid process is applicable for C programs. However, the case studies show there are still some drawbacks that can be used to evaluate and improve the Robustness Grid measurement process.

Furthermore, the case studies show that the robustness measurement results fairly reflect the accuracy of the program syntax writing, where the Robustness Grid highlights the problems that are expected to be caught, in addition to some other problems.

The evaluation of the Robustness Grid is divided into three parts and described in Chapter 6. The first part is the general Robustness Grid critique, depending on the SwapoAdd.c case study results. The results show that the Robustness Grid is a

measurement technique that presents the satisfied rules as well as the violated ones, which gives a clue to the maintainer on what needs to be fixed. It also has a flexible way of presenting results and from different points of views. The Clause Slicing technique helps discover the importance and the Clause of the different piece of code in the program.

In Section 6.4, the second part of the evaluation process is based on the comparison between the Robustness Grid and other Robustness measurement techniques. The comparison is made with tools that use MISRA C2 rules such as: LDRA TBmisra, FlexeLint, and Klocwork Truepath. The comparison shows that the Robustness Grid provides services that none of the previous tools provide. On the other hand, the previous tools support more rules than the Robustness Grid. The comparison shows that MISRA C2 were interpreted and applied differently in the tools including Robustness Grid, therefore, the results of each tool measurement is different from the other.

Besides the previous tools, the Robustness Grid is compared with the fault injection technique using program slicing that was introduced in the Gallagher and Fulton paper (Gallagher and Fulton 1999). Even though both of the Robustness Grid and fault injection used slicing to analyse the program code, they measured the Robustness Degree from a different perspective. The Robustness Grid measures the program by assessing the code syntax. The fault injection technique measures the robustness using the input/output relation.

The third part of the evaluation described in Section 6.5, shows a case study, The Robost.c, which gives an idea of how many rules are needed to build the smallest robust program from scratch. The Robost.c is introduced as a base program that can be used to build a robust program. The code lines can be added to Robost.c after they are tested and satisfied the MISRA C2 rules. Subsequently, a 100% robust program can be created.

The results of the analysis and evaluation show that Robustness Grid model is capable of producing a sufficient Robustness Degree measurement for the programs written in the C language.

The Robustness Grid helps the developer team to understand the program robustness measurement results. The Managerial-View helps the managers to understand the measurement results in the big picture. The different tables that build the Robustness Grid give a detailed analysis for the measurement results.

For example, the Managerial-View gives the manager an indication about which function and category are the most important in the program, in addition to which function and category have the least Robustness Degree value.

The Clause Table, shows the importance level of the program clauses and which rules that has been applied and whether satisfied or violated in the program. The Data Table shows the importance of the Rules that has been applied in the program and their satisfaction status.

7.3 Criteria for Success

The criteria for success of the research in this thesis were presented in Chapter 1. This section discusses the achievement of these criteria. These achievements are as follows:

1- Develop a measurement for assessing the Robustness of C programs

   In this thesis, the developed measurement is made using the MISRA C2 rules with Clause Slicing. This was achieved by analysing the C program and then assessing the program clauses using the MISRA C2 rules and weighting them using the Clause Slicing technique in the new Robustness measurement technique introduced in this research. In Chapter 3, the

proposed model was introduced, where in Chapter 5; the main case study was discussed in detail.

2- Develop a Grid that incorporates the robustness measurement

The Robustness Grid is the proposed model introduced in this research and discussed through this thesis. It is a robustness measurement framework that uses the MISRA C2 rules and the Clause Slicing technique.

The Robustness Grid is a table that presents function and rule robustness measurements. The Robustness Grid is built depending on two tables; the Clause Table which was discussed in Section 3.4.2.1 and Data Table in Section 3.4.2.2. The Robustness Grid is the combined of the Clause and Data Tables with some changes in one big table containing the Function Category robustness Degree described in Section 3.4.5 and the Category Calculations described in Section 3.5.2.

3- Empirically evaluate the Grid

The evaluation of the Robustness Grid was made in three phases; the first is the evaluation of the Robustness Grid depending on the results of the main case study introduced in Chapter 5. The second part of the evaluation is a comparison with existing tools and techniques that measure or assess the program robustness, which was described in Chapter 6 Section 6.4 and Subsection 6.5.1.

The third part is an evaluation of the ability and consistency of the Robustness Grid measurement for random C programs, by introducing two more case studies in Subsections 6.5.1 and 6.5.2, and building a robust program from scratch.

4- Compare the results against other related studies

The Robustness Grid was compared with three different techniques that use MISRA C2 language rules to measure the program robustness, and one case study taken from the Gallagher and Fulton paper (Gallagher and Fulton 1999), that use program slicing to measure the program robustness. Comparison discussions and tables were described in Sections 6.4 and 6.5.1.

5- Develop a proof of concept of implementation

This study has produced parts of the Robustness Grid in prototype mode. As described in Chapter 4, the Robustness Grid is not a fully automated model. However, some phases of the model used suitable existing tools. The program slicing technique was executed by CodeSurfer (CSurf) and the Robustness Grid Calculations and display were done using Microsoft Excel sheets (MS Excel).

7.4 Future Directions

Although the proposed model in this thesis has considerably achieved the intended goals, there are still some possible additions that can be done to enhance it. These additions are as follows:

1- Fully automated and repeatable

The Robustness Grid uses existing tools to implement part of the program analysis and robustness measurements. Therefore, in future research, all phases of the Robustness Grid will be fully implemented. Then, it can become fully integrated and accomplished tool.

2- Cover all MISRA C2 rules

The Robustness Grid only uses 100 out of 142 MISRA C2 rules in the robustness measurement. The future direction is to make the Robustness Grid able to measure all aspects addressed by all the MISRA C2 rules.

3- Improving generality of the model

The Robustness Grid only measures selected C programs executable through a C compiler. Thus, in the future, the Robustness Grid needs to be able to measure any C programs, or any part of it.

4- Evaluating the Robustness Grid using large programs

Since the Robustness Grid is not fully automated, only small programs are used to analyse and evaluate the Robustness Grid. In future research, bearing in mind the previous suggestions, the Robustness Grid can be evaluated using large C programs.

5- Evaluate the Robustness Grid using different language standard

The current Robustness Grid uses the MISRA C2 rules to measure the C programs. In the future, different standards could be used to measure C programs or programs of a different programming language.

6- Build a 100% robust program

The evaluation section, 6.5.3, introduced a 100% robust program, the Robost.c program. This program is the smallest robust program, but it does nothing. In the future, Robost.c can be used as a base to build different robust programs. This issue was raised in a paper published in 2010 (Abdallah, Munro et al. 2010).

7- Implement the Robustness Grid in real life applications

The Robustness Grid can help the developer and maintainer because it gives them an indication of the weak robustness instances in a piece of code, functions, and category that need to be maintained to improve the robustness degree.

7.5 Summary

This thesis has discussed the research into program robustness measurement. In this research, a new robustness measurement technique was introduced, called the Robustness Grid. The Robustness Grid uses the language standards rules and the program slicing technique to produce a detailed robustness measurement for a program and its functions.

This research shows evidence that the Robustness Grid can produce a precise measurement for C programs using the MISRA C2 rules and the Clause Slicing technique.

References

Abdallah, M., M. Munro and K. Gallagher (2010). Certifying software robustness using program slicing. IEEE International Conference on Software Maintenance. Timisoara, Romania**:** 1-2.

Abdallah, M., M. Munro and K. Gallagher (2011). A Static Robustness Grid Using MISRA C2 Language Rules. The Sixth International Conference on Software Engineering Advances, ICSEA 2011. Barcelona, Spain**:** 65-69.

Agrawal, H., R. A. Demillo and E. H. Spafford (1993). Debugging with dynamic slicing and backtracking, John Wiley and Sons, Inc. **23:** 589-616.

Agrawal, H. and J. R. Horgan (1990). Dynamic program slicing. Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation. New York, United States, ACM. **25:** 246-256.

Agrawal, H., J. R. Horgan, E. W. Krauser and S. London (1993). Incremental Regression Testing. Proceedings of the Conference on Software Maintenance, IEEE Computer Society**:** 348-357.

American National Standards Institute (ANSI). (1999). "http://webstore.ansi.org/RecordDetail.aspx?sku=ISO/IEC+9899:1999." Retrieved 20/3/2012, 2012.

Arup, M. and P. S. Daniel (1997). Measuring Software Dependability by Robustness Benchmarking, IEEE Press. **23:** 366-378.

Avizienis, A., J. C. Laprie, B. Randell and C. Landwehr (2004). "Basic Concepts and Taxonomy of Dependable and Secure Computing." IEEE Transactions on Dependable and Secure Computing **1**(1): 11-33.

Ayewah, N., D. Hovemeyer, J. D. Morgenthaler, J. Penix and W. Pugh (2008). "Using Static Analysis to Find Bugs." IEEE Transactions on Software Engineering **25**(5): 22-29.

Baowen, X., Q. Ju, Z. Xiaofang, W. Zhongqiang and C. Lin (2005). "A Brief Survey of Program Slicing." ACM SIGSOFT Software Engineering Notices **30**(2): 1-36.

Binkley, D. (1998). "The application of program slicing to regression testing." Information and Software Technology **40**(11-12): 583-594.

Binkley, D., S. Danicic, G. Tibor, thy, M. Harman, K. kos and K. Bogdan (2006). A Formalisation of the Relationship Between Forms of Program Slicing, Elsevier North-Holland, Inc. **62:** 228-252.

Binkley, D. and K. Gallagher (1996). Program Slicing. Advances in Computers. V. Z. Marvin, Elsevier. **43:** 1-50.

Binkley, D. and M. Harman (2005). Locating Dependence Clusters and Dependence Pollution. Proceedings of the 21st IEEE International Conference on Software Maintenance, IEEE Computer Society**:** 177-186.

Black, S. (2001). "Computing Ripple Effect for Software Maintenance." Journal of Software Maintenance **13**(4): 263.

Brown, A. B., J. Hellerstein, M. Hogstrom, T. Lau, S. Lightstone, P. Shum and M. P. Yost (2004). Benchmarking Autonomic Capabilities: Promises and Pitfalls. International Conference on Autonomic Computing (ICAC'04). Los Alamitos, CA, USA, IEEE Computer Society**:** 266-267.

Brown, A. B. and C. Redlin (2005). Measuring the Effectiveness of Self-Healing Autonomic Systems. Proceedings of the Second International Conference on Autonomic Computing, 2005. ICAC 2005.**:** 328-329.

Canfora, G., A. Cimitile and A. DeLucia (1998). "Conditioned program slicing." Information and Software Technology **40**(11-12): 595-607.

Chung, I. S., W. K. Lee, G. S. Yoon and Y. R. Kwon (2001). Program slicing based on specification. Proceedings of the 2001 ACM symposium on Applied computing. Las Vegas, Nevada, United States, ACM**:** 605-609.

Dabek, F., N. Zeldovich, F. Kaashoek, D. Mazi and R. Morris (2002). Event-driven programming for robust software. Proceedings of the 10th workshop on ACM SIGOPS European workshop. Saint-Emilion, France, ACM**:** 186-189.

Danicic, S., C. Fox, M. Harman and R. Hierons (2000). ConSIT: A Conditioned Program Slicer. IEEE International Conference on Software Maintenance (ICSM'00), IEEE Computer Society Press**:** 216-226.

DeLucia, A. (2001). "Program slicing: methods and applications." Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation, 2001. : 144-151.

DeLucia, A. (2001). Program Slicing: Methods and Applications. IEEE International Workshop on Source Code Analysis and Manipulation.

DeLucia, A., A. R. Fasolino and M. Munro (1996). Understanding Function Behaviors through Program Slicing. 4th International Workshop on Program Comprehension. Berlin, Germany, IEEE Computer Society**:** 9 -18.

DeVale, J. and P. J. Koopman (2002). Robust Software - No More Excuses. Proceedings of the 2002 International Conference on Dependable Systems and Networks, IEEE Computer Society**:** 145-154.

DeVale, J. P., P. J. Koopman and D. J. Guttendorf (1999). The Ballista Software Robustness Testing Service. Tesing Computer Software Coference.

Dix, M. and H. D. Hofmann (2002). Automated software robustness testing - static and adaptive test case design methods. Proceedings of 28th Euromicro Conference**:** 62-66.

Drexel University. (2012). "C Language Tutorial." Retrieved 31/3/2012, from http://einstein.drexel.edu/courses/Comp_Phys/General/C_basics/#arrays.

Eslamnour, B. and S. Ali (2009). "Measuring Robustness of Computing Systems." Simulation Modelling Practice and Theory **17**(9): 1457-1467.

Fatiregun, D., M. Harman and R. M. Hierons (2005). Search-Based Amorphous Slicing. Proceedings of the 12th Working Conference on Reverse Engineering, IEEE Computer Society**:** 3-12.

Fenton, N. E. and S. L. Pfleeger (1997). Software Metrics, A Rigorous and Practical Approach, PWS Publishing Company.

Gallagher, K. and D. Binkley (2008). Program slicing. Frontiers of Software Maintenance, FoSM 2008.**:** 58-67.

Gallagher, K. and N. Fulton (1999). Using Program Slicing to Estimate Software Robustness. Proceedings of the International Systems Software Assurance Conference, ISSAC.

Gallagher, K., T. Hall and S. Black (2007). Reducing Regression Test Size by Exclusion. IEEE International Conference on Software Maintenance, 2007. ICSM**:** 154-163.

Gallagher, K. and J. R. Lyle (1991). "Using Program Slicing in Software Maintenance." IEEE Transactions on Software Engineering **17**(8): 751-761.

Gallagher, K. and L. O'Brien (2001). Analyzing Programs via Decomposition Slicing Initial Data and Observation. Proceeding of 7th workshop on Empirical Studies of Software Maintenance, Florence, Italy.

Gallagher, K. B. and J. R. Lyle. (1998). "The Surrgeon's Assistant." Retrieved 30/3/2012, from http://www.cs.loyola.edu/~kbg/Surgeon/.

Gimple Software. (2012). "FlexeLint." Retrieved 30/3/2012, from http://www.gimpel.com/html/index.htm.

GrammaTech. (2012). "CodeSurfer." Retrieved 25/3/2012, 2012, from http://www.grammatech.com/products/codesurfer/overview.html.

Gramoli, V., Y. Vigfusson, K. Birman, A.-M. Kermarrec and R. v. Renesse (1999). "Slicing Distributed Systems." IEEE Transactions on Computers **99**(1).

Gribble, S. D. (2001). Robustness in Complex Systems. Proceedings of the Eighth Workshop on Hot Topics in Operating Systems, IEEE Computer Society**:** 21-26.

Hall, R. J. (1995). "Automatic extraction of executable program subsets by simultaneous dynamic program slicing." Automated Software Engineering **2**(1): 33-53.

Hamann, A., R. Racu and R. Ernst (2007). Methods for multi-dimensional robustness optimization in complex embedded systems. Proceedings of the 7th ACM & IEEE international conference on Embedded software. Salzburg, Austria, ACM**:** 104-113.

Harman, M., D. Binkley and S. Danicic (2003). "Amorphous program slicing." Journal of Systems and Software **68**(1): 45-64.

Harman, M. and S. Danicic (1995). "Using Program Slicing to Simplify Testing." Journal of Software Testing, Verification and Reliability **5**(3): 143-162.

Harman, M. and S. Danicic (1997). Amorphous Program Slicing. International Workshop on Program Comprehension, WPC. Dearborn, MI, USA, IEEE Computer Society**:** 70-79.

Harman, M. and R. Hierons (2001). "An Overview of program slicing." software focus **2**(3): 85-92.

Hatcliff, J., M. B. Dwyer and H. Zheng (2000). Slicing Software for Model Construction, Kluwer Academic Publishers. **13:** 315-353.

Horwitz, S., T. Reps and D. Binkley (1990). "Interprocedural slicing using dependence graphs." ACM Transaction of Program Language Systms **12**(1): 26–60.

Huhns, M., V. Holderfield and R. Gutierrez (2003). Robust Software via Agent-based Redundancy. Proceedings of the second international joint conference on Autonomous agents and multiagent systems. Melbourne, Australia, ACM**:** 1018-1019.

Huhns, M. N. and V. T. Holderfield (2002). "Robust Software." IEEE Internet Computing **6**(2): 80-82.

IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990, IEEE Computer Society.

ISO. (2012). "International Organization for Standardisation." Retrieved 20/3/2012, 2012, from http://www.iso.org/iso/home.html.

ISO/IEC (1999). International Standard ISO/IEC 9899, International Organaization for Standardization.

ISO/IEC (2007). ISO/IEC 15939: Systems and software engineering -- Measurement process. ISO/IEC.

Jackson, D. and E. J. Rollins (1994). Chopping: A Generalization of Slicing, Carnegie Mellon University.

Jawadekar, W. S. (2004). Software Engineering: Principles and Practice, Mcgraw Hill Higher Education.

Jones, D. M. (2009). The New C Standard: A Cultural and Economic Commentary, Addison-Wesley Professional.

Junhua, W., X. Baowen and J. Jixiang (2004). Slicing Web Application Based on Hyper Graph. Proceedings of the 2004 International Conference on Cyberworlds, IEEE Computer Society**:** 177-181.

Kanoun, K., H. Madeira and J. Arlat (2002). A Framework for Dependability Benchmarking. The International Conference on Dependable Systems and Networks. Washington, D.C., USA**:** 7-8.

Kaur, K., K. Minhas, N. Mehan and N. Kakkar (2009). "Static and Dynamic Complexity Analysis of Software Metrics." Empirical Software Engineering **56**(V): 159-161.

Kim, D. and M. Fong (2007). Research Report: Program Slicing.

Klocwork. (2012). "Detected MISRA Issues." Retrieved 29/3/2012, from http://www.klocwork.com/products/documentation/current/current:Books/Detected_MISRA_Issues.

Klocwork. (2012). "Klocwork Insight." Retrieved 20/3/2012, from http://www.klocwork.com/products/insight/.

Koopman, P. (2002). "The Ballista Project: COTS Software Robustness Testing." Retrieved 30/3/2012, 2012, from http://www.ece.cmu.edu/~koopman/ballista/index.html.

Koopman, P., K. Devale and J. Devale (2008). Interface Robustness Testing: Experience and Lessons Learned from the Ballista Project. Dependability Benchmarking for Computer Systems, John Wiley & Sons, Inc.**:** 201-226.

Korel, B. and J. Laski (1990). "Dynamic slicing of computer programs." Journal of Systems and Software **13**(3): 187-195.

Korel, B. and J. Rilling (1998). "Dynamic program slicing methods." Information and Software Technology **40**(11-12): 647-659.

Krinke, J. (2004). Slicing, Chopping, and Path Conditions with Barriers, Kluwer Academic Publishers. **12:** 339-360.

Laddaga, R. (1999). Guest Editor's Introduction: Creating Robust Software through Self-Adaptation. **14:** 26-29.

Lalchandani, J. T. and R. Mall (2008). Regression testing based-on slicing of component-based software architectures. Proceedings of the 1st conference on India software engineering conference. Hyderabad, India, ACM**:** 67-76.

Laprie, J. C., J. Arlat, C. Beounes and K. Kanoun (1990). "Definition and analysis of hardware- and software-fault-tolerant architectures." Computer **23**(7): 39-51.

Laroche, E. (1998). "C programming language coding guidelines." Retrieved 30/3/2012, from http://www.lrdev.com/lr/c/ccgl.html.

Larsen, L. and M. J. Harrold (1996). Slicing object-oriented software. Proceedings of the 18th international conference on Software engineering. Berlin, Germany, IEEE Computer Society**:** 495-505.

LDRA. (2012). "Dynamic Analysis with LDRA Testbed®." Retrieved 30/3/2012, from http://www.ldra.com/dynamicanalysis.asp.

LDRA. (2012). "LDRA products and services." Retrieved 20/3/2012, 2012, from http://www.ldra.com/products.asp.

LDRA. (2012). "Static Analysis with LDRA Testbed®." Retrieved 30/3/2012, from http://www.ldra.com/staticanalysis.asp.

Lyu, M. R., H. Zubin, S. K. S. Sze and C. Xia (2003). An empirical study on testing and fault tolerance for software reliability engineering. 14th International Symposium on Software Reliability Engineering, ISSRE 119-130.

Maguire, S. (1993). Writing Solid Code. Washington, Microsoft Press.

Maule, A., W. Emmerich and D. S. Rosenblum (2008). Impact analysis of database schema changes. Proceedings of the 30th international conference on Software engineering. Leipzig, Germany, ACM**:** 451-460.

Mazeiar, S. and T. Ladan (2009). "Self-adaptive software: Landscape and research challenges." ACM Trans. Auton. Adapt. Syst. **4**(2): 1-42.

Mieczyslaw, M. K., B. Kenneth and A. E. Yonet (1999). Control Theory-Based Foundations of Self-Controlling Software, IEEE Educational Activities Department. **14:** 37-45.

Miller, B. P., D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan and J. Steidl (1995). Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services, University of Wisconsin-Madison, Computer Sciences Dept. **1525:** 1-23.

MISRA (2004). MISRA-C: 2004 Guidlines for the use of the C Language in Critical Systems. The Motor Industry Software Reliability Association, MIRA limited.

MISRA. (2012). "MISRA's Mission Statement." Retrieved 20/3/2012, from http://www.misra.org.uk/MISRAHome/WhatisMISRA/tabid/66/Default.aspx.

Musa, J. D., A. Iannino and K. Okumoto (1987). Software Reliability: measurement, prediction, application, McGraw-Hill, Inc.

Ngah, A. and K. Gallagher (2009). Regression test selection by exclusion using decomposition slicing. Proceedings of the doctoral symposium for ESEC/FSE on Doctoral symposium. Amsterdam, The Netherlands, ACM**:** 23-24.

Petitpierre, C. and A. Eliëns (2002). Active Objects Provide Robust Event-Driven Applications. SERP'02. H. R. A. a. Y. Mun. Las Vegas**:** 253-259.

Philip, G. C. (1998). "Software design guidelines for event-driven programming." **41**(2): 79-91.

Pressman, R. S. (2009). Software Engineering: A Practitioner's Approach, McGraw Hill Higher Education.

Pullum, L. L. (2001). Software Fault Tolerance Techniques and Implementation, Artech House, Inc.

Qian, J. and B. Xu (2008). "Program slicing under UML scenario models." ACM SIGPLAN Notices **43**(2): 21-24.

Ranganath, V. P. and J. Hatcliff (2007). Slicing concurrent Java programs using Indus and Kaveri, Springer-Verlag. **9:** 489-504.

Rebaudengo, M., M. Sonza Reorda, M. Torchiano and M. Violante (1999). Soft-error detection through software fault-tolerance techniques. International Symposium on Defect and Fault Tolerance in VLSI Systems, DFT '99**:** 210-218.

Sasirekh, N., A. EdwinRober and M. Hemalth (2011). "Program slicing techniques and its applications." International Journal of Software Engineering & Applications (IJSEA) **2**(3): 50-64.

Schmid, M. and F. Hill (1999). Data generation techniques for automated software robustness testing. Proceedings of the International Conference on Testing Computer Software**:** 14-18.

Seacord, R. (2012). "CERT C Secure Coding Standard." Retrieved 30/3/2012, 30/3/2012, from https://www.securecoding.cert.org/confluence/display/seccode/CERT+C+Secure+Coding+Standard.

Shahrokni, A. and R. Feldt (2010). Towards a Framework for Specifying Software Robustness Requirements Based on Patterns, Requirements engineering: Foundation for software quality**:** 79-84.

Shahrokni, A. and R. Feldt (2011). RobusTest: Towards a Framework for Automated Testing of Robustness in Software. VALID 2011: The Third International Conference on Advances in System Testing and Validation Lifecycle. Barcelona, Spain**:** 78-83.

Shinji, K., N. Akira, N. Keisuke and I. Katsuro (2002). Experimental Evaluation of Program Slicing for Fault Localization, Kluwer Academic Publishers. **7:** 49-76.

Software Engineering Institute. (2011). "CERT C Secure Coding Standard." Retrieved 30/3/2012, 30/3/2012, from https://www.securecoding.cert.org/confluence/display/seccode/CERT+C+Secure+Coding+Standard.

Sommerville, I. (2008). Software Engineering, Addison-Wesley.

Straker, D. (1992). C Style: Standards and Guidlines, Defining Programming Standards for Professional C programmers, Prentice Hall International.

Thompson, M. (2010). "CERT C - Risk Assessment." Retrieved 29/3/2012, from https://www.securecoding.cert.org/confluence/display/seccode/Risk+Assessment.

Tip, F. (1995). "A survey of Program Slicing Techniques." Journal of Programming Languages **3**: 121-189.

Tonella, P. (2003). "Using a concept lattice of decomposition slices for program understanding and impact analysis." IEEE Transactions on Software Engineering **29**(6): 495-509.

Tonella, P. and F. Ricca (2005). "Web Application Slicing in Presence of Dynamic Code Generation." Automated Software Engineering **12**(2): 259-288.

Venkatesh, G. A. (1991). The semantic approach to program slicing, ACM. **26:** 107-119.

Voas, E., F. Charron, G. McGraw, K. Miller and M. Friedman (1997). "Predicting how badly "good" software can behave." Software, IEEE **14**(4): 73-83.

Wang, T. and A. Roychoudhury (2004). Using compressed bytecode traces for slicing Java programs. ACM/IEEE International Conference on Software Engineering (ICSE)**:** 512-521.

Weinberg, G. M. (1983). Kill That Code! Infosystems. **8:** 48-49.

Weiser, M. (1979). Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. Michigan, The University of Michigan. **PhD:** 270.

Weiser, M. (1981). Program Slicing. Proceedings of the 5th international conference on Software Engineering. California, United States, IEEE Press**:** 439-449.

Weiser, M. (1982). "Programmers use slices when debugging." Communications of the ACM **25**(7): 446-452.

Weiser, M. (1984). "Program Slicing." IEEE Transactions on Software Engineering **10**: 352-357.

Weiser, M. and J. Lyle (1986). Experiments on slicing-based debugging aids. Papers presented at the first workshop on Empirical studies of programmers. Washington, D.C., United States, Ablex Publishing Corp**:** 187-197.

Zhao, J. (2002). Slicing aspect-oriented software. Program Comprehension, 2002. Proceedings. 10th International Workshop on**:** 251-260.

Zuse, H. (1998). A Framework of Software Measurement, Walter de Gruyter.

Appendices

Appendix A: Rule Distribution in the Robustness Grid

Category 1	Category 2	Category 3	Category 4	Category 5	Category 6
4.1+7.1	12.2	8.1	8.12	8.4	5.7
4.2	12.3	8.2	9.2	19.4	6.3
5.1	12.4	8.3	11.5	19.5	11.3
5.2	12.5	8.6	16.7	19.6	11.4
6.1	12.7	8.7		19.9	12.1
6.2	12.8	8.8		19.10	12.6
6.4	13.1	8.11		19.11	12.11
6.5	13.3	14.8		19.(12+13)	13.2
10.1	13.4	16.1		19.15	19.1
10.2	13.5	16.2		19.16	19.2
10.3	13.6	16.3		20.1	19.7
10.4	13.7	16.4		20.2	
10.5	14.4	16.5		20.3	
10.6	14.5	16.8		20.4	
12.9	14.7	16.9		20.6	
12.10	14.9			20.7	
12.12	14.10			20.8	
12.13	15.1			20.9	
14.3	15.2			20.10	
	15.3			20.11	
	15.4			20.12	
	15.5				
	17.1				
	17.2				
	17.3				
	17.4				
	17.5				
	17.6				

Rules distribution on their Categories

Appendix B: SwapoAdd.c program

```
#include <stdio.h>1
#define LAST 102
#define ARRAY_SIZE 103
typedef unsigned char x1;4
char array[ARRAY_SIZE] = "0123456789";5
void incr(int *num, int i);6
void swap(int *a, int *b);7
int one (int x);8
int main()9
{int i;10
int sum = 0;11
int  *a = 12; 12
int *b = 13;13
int  x = 3;14
for ( i = 115; i <= LAST16; i++ 17) {
incr18(&sum19, i20); }
printf21("sum = %d\n"22, sum23);
swap24 (&a25,&b26);
int index27;
for (index = 028; index < ARRAY_SIZE29; ++index30)
{one31 (x32);
printf33("&array[index]=0x%p (array+index)=0x%p array[index]=0x%x\n"34,
&array[index],35
 (array+index),36
array[index])37;}
return 0;}38
void incr39(int *num40, int i41) {
*num= *num + i;}42
void swap43(int *a44, int *b45){
int temp = *a46;
*a = *b47;
*b = temp48;
printf49 ("pointer a is: %d\n"50,*a51);
printf 52("pointer b is: %d\n"53,*b54);}
int one55 (int x56)
{printf57 ("%d\n"58, x59);
return 1;60}
```

SwapoAdd.c Program with the Clauses Numbers

Appendix C: Clause Table of SwapoAdd.c program

Clause Number	Slice Size	Clause Frequency	Clause Weight	Function Name	Applicable Rules	
					Satisfied	Violated
1*	1	1	1	main	19.1, 19.2,20.9, 20.2,20.1	0
2*	1	1	1	main	19.6	0
3*	1	1	1	main	19.6	0
4*	1	1	1	main	0	6.3
5	4	2	8	main	5.1, 6.1, 8.12, 13.1, 9.2	0
6*	1	1	1	incr	16.3, 16.4, 16.1	19.7
7*	1	1	1	swap	16.3, 16.4, 16.1	19.7
8*	1	1	1	one	16.3, 16.4, 16.1	19.7
9	53	2	106	main	14.7, 16.1, 8.2, 8.6	16.5, 19.7, 8.1
10	12	2	24	main	5.1	0
11	5	2	10	main	5.1	0
12	8	2	16	main	5.1, 17.5	0
13	7	2	14	main	5.1, 17.5	0
14	4	2	8	main	5.1	0
15	11	3	33	main	13.4, 13.5, 14.8	0
16	10	5	50	main	13.5, 13.4	0
17	10	5	50	main	12.3, 13.5, 13.4	0
18	8	6	48	incr	16.2, 16.9	13.6
19	4	7	28	incr	0	0
20	4	7	28	incr	0	0
21	3	2	6	main	16.2, 16.9	16.1
22	1	3	3	main	4.1, 4.2	0
23	1	15	15	main	0	0
24	15	2	30	swap	16.2,16.9	0
25	7	4	28	swap	0	0
26	6	4	24	swap	0	0
27	17	2	34	main	5.1	0
28	16	3	48	main	13.5, 14.8,13.4	0
29	15	5	75	main	13.5, 16.2, 13.4	0
30	15	5	75	main	12.3, 13.5, 13.4	0
31	8	6	48	one	16.2, 16.9	0
32	3	8	24	one	0	0
33	5	6	30	main	16.9, 16.2	0
34	1	7	7	main	4.1, 4.2, 12.1,12.2, 17.4, 17.4	7.1, 7.1, 7.1
35	1	8	8	main	0	0
36	1	8	8	main	12.1, 12.2	0

Clause Number	Slice Size	Clause Frequency	Clause Weight	Function Name	Applicable Rules	
					Satisfied	Violated
37	1	8	8	main	0	0
38	2	2	4	main	16.8	0
39	5	7	35	incr	8.1, 8.2, 8.6	14.7
40	3	10	30	incr	5.1, 8.3	0
41	3	10	30	incr	5.1, 8.3	0
42	2	12	24	incr	12.2	17.1, 17.4
43	12	3	36	swap	8.1, 8.2, 8.6	14.7
44	6	7	42	swap	5.1, 8.3	5.2
45	5	7	35	swap	5.1, 8.3	5.2
46	3	7	21	swap	5.1	11.3
47	2	10	20	swap	0	0
48	2	11	22	swap	0	11.3
49	3	4	12	swap	16.2, 16.9	16.1
50	1	5	5	swap	4.1, 4.2	0
51	1	12	12	swap	0	0
52	3	4	12	swap	16.2, 16.9	16.1
53	1	5	5	swap	4.1, 4.2	0
54	1	13	13	swap		0
55	6	8	48	one	8.1, 8.2, 8.6, 14.7	0
56	2	10	20	one	5.1, 8.3	5.2
57	3	8	24	one	16.2, 16.9	16.1
58	1	9	9	one	4.1, 4.2	0
59	1	12	12	one	0	0
60	2	8	16	one	16.8	0
* Un-sliceable Clauses						

Clause Table of SwapoAdd.c program

Appendix D: Data Table of SwapoAdd.c program

Rule Number	Number of Satisfied	∑Satisfied Slices	Number of Violated	∑Violated Slices
4.1 + 7.1	5	5	3	3
4.2	5	5	0	0
5.1	13	79	0	0
5.2	0	0	3	13
6.1	1	4	0	0
6.3	0	0	1	1
8.1	3	23	1	53
8.2	4	76	0	0
8.3	5	19	0	0
8.6	4	75	0	0
8.12	1	4	0	0
9.2	1	4	0	0
11.3	0	0	2	5
12.1	2	2	0	0
12.2	3	3	0	0
12.3	2	25	0	0
13.4	6	77	0	0
13.5	6	77	0	0
13.6	0	0	1	8
14.7	2	59	2	17
14.8	2	27	0	0
16.1	4	56	4	10
16.2	9	63	0	0
16.3	3	3	0	0
16.4	3	3	0	0
16.5	0	0	1	53
16.8	2	4	0	0
16.9	8	48	0	0
17.1	0	0	1	2
17.4	2	2	1	2
17.5	2	15	0	0
19.1	1	1	0	0
19.2	1	1	0	0
19.6	2	2	0	0
19.7	0	0	4	56
20.2	1	1	0	0
20.9	1	1	0	0

Data Table of SwapoAdd.c program

Appendix E: *swap* Function Calculation Table

Categories	Rule Number	*swap*								FCD %	
		Applied Rules	∑Satisfied Slice Sizes	Satisfied Weight	∑Violated Slice Sizes	Violated Weight	Function Frequency	Rule ∑ Function Slice Size	Rule Function Weight	FCSD %	FCVD %
Category 1	4.1 + 7.1	+2	2	4	0	0	2	2	4	7/9 = 77.78%	2/9= 22.22%
	4.2	+2	2	4	0	0	2	2	4		
	5.1	+3	14	42	0	0	3	14	42		
	5.2	-2	0	0	11	22	2	11	22		
	6.1	0	0	0	0	0	0	0	0		
AC	5	4	18	50	11	22	9	29	72	7/9 = 77.78%	2/9= 22.22%
Category 2	12.2	0	0	0	0	0	0	0	0	0/1 = 0%	1/1 = 100%
	12.3	0	0	0	0	0	0	0	0		
	13.4	0	0	0	0	0	0	0	0		
	13.5	0	0	0	0	0	0	0	0		
	13.6	0	0	0	0	0	0	0	0		
	14.7	-1	0	0	12	12	1	12	12		
	17.1	0	0	0	0	0	0	0	0		
	17.5	0	0	0	0	0	0	0	0		
AC 0-1	13	5	18	100	23	56	10	41	84	7/10= 70%	3/10= 30%
Category 3	8.1	+1	12	12	0	0	1	12	12	14/16 = 87.5%	2/16 = 12.5%
	8.2	+1	12	12	0	0	1	12	12		
	8.3	+2	11	22	0	0	2	22	44		
	8.6	+1	12	12	0	0	1	12	12		
	14.8	0	0	0	0	0	0	0	0		
	16.1	+1/-2	1	1	6	12	3	7	21		
	16.2	+3	21	63	0	0	3	21	63		
	16.3	+1	1	1	0	0	1	1	1		
	16.4	+1	1	1	0	0	1	1	1		
	16.5	0	0	0	0	0	0	0	0		
	16.8	0	0	0	0	0	0	0	0		
	16.9	+3	21	51	0	0	3	51	153		
AC 0-2	25	14	92	275	6	68	26	180	403	21/26= 80.76%	5/26= 19.24%
Category 4	8.12	0	0	0	0	0	0	0	0	0	0
	9.2	0	0	0	0	0	0	0	0		
AC 0-3	27	14	92	275	0	68	26	180	403	21/26= 80.76%	5/26= 19.24%
Category 5	19.6	0	0	0	0	0	0	0	0	0	0
	20.2	0	0	0	0	0	0	0	0		
	20.9	0	0	0	0	0	0	0	0		
AC 0-4	30	14	92	275	0	68	26	180	403	21/26= 80.76%	5/26= 19.24%

Categories	Rule Number	*swap*								FCD %	
		Applied Rules	∑Satisfied Slice Sizes	Satisfied Weight	∑Violated Slice Sizes	Violated Weight	Function Frequency	Rule ∑ Function Slice Size	Rule Function Weight	FCSD%	FCVD%
Category 6	6.3	0	0	0	0	0	0	0	0	0/3 = 0%	3/3 = 100%
	11.3	-2	0	0	5	10	2	5	10		
	12.1	0	0	0	0	0	0	0	0		
	17.4	0	0	0	0	0	0	0	0		
	19.1	0	0	0	0	0	0	0	0		
	19.2	0	0	0	0	0	0	0	0		
	19.7	-1	0	0	1	1	1	1	1		
FAC	37	16	92	275	6	79	29	186	414	21/29 = 72.41%	8/29= 27.59%

***swap* Function Calculation**

Appendix F: *one* Function Calculation Table

Categories	Rule Number	*one*								FCD %	
		Applied Rules	∑Satisfied Slice Sizes	Satisfied Weight	∑Violated Slice Sizes	Violated Weight	Function Frequency	Rule ∑ Function Slice Size	Rule Function Weight	FCSD %	FCVD %
Category 1	4.1 + 7.1	+1	1	1	0	0	1	1	1	3/4 = 75%	1/4 = 25%
	4.2	+1	1	1	0	0	1	1	1		
	5.1	+1	2	2	0	0	1	2	2		
	5.2	-1	0	0	2	2	1	2	2		
	6.1	0	0	0	0	0	0	0	0		
AC	5	4	4	4	2	2	4	6	6	3/4 = 75%	1/4 = 25%
Category 2	12.2	0	0	0	0	0	0	0	0	1/1 = 100%	0/1 = 100%
	12.3	0	0	0	0	0	0	0	0		
	13.4	0	0	0	0	0	0	0	0		
	13.5	0	0	0	0	0	0	0	0		
	13.6	0	0	0	0	0	0	0	0		
	14.7	+1	6	6	0	0	1	6	6		
	17.1	0	0	0	0	0	0	0	0		
	17.5	0	0	0	0	0	0	0	0		
AC 1-2	13	5	10	14	2	2	5	12	12	4/5 = 80%	1/5 = 20%
Category 3	8.1	+1	6	6	0	0	1	6	6	12/13 = 92.3%	1/13 = 7.7%
	8.2	+1	5	5	0	0	1	5	5		
	8.3	+1	2	2	0	0	1	2	2		
	8.6	+1	6	6	0	0	1	6	6		
	14.8	0	0	0	0	0	0	0	0		
	16.1	+1/-1	1	1	3	3	2	4	8		
	16.2	+2	11	22	0	0	2	11	22		
	16.3	+1	1	1	0	0	1	1	1		
	16.4	+1	1	1	0	0	1	1	1		
	16.5	0	0	0	0	0	0	0	0		
	16.8	+1	2	2	0	0	1	2	2		
	16.9	+2	11	22	0	0	2	11	22		
AC 1-3	25	15	46	82	5	5	18	61	87	16/18 = 88.88%	2/18 =11.12%
Category 4	8.12	0	0	0	0	0	0	0	0	0	0
	9.2	0	0	0	0	0	0	0	0		
AC 1-4	27	15	46	82	5	5	18	61	87	16/18 = 88.88%	2/18 =11.12%
Category 5	19.6	0	0	0	0	0	0	0	0	0	0
	20.2	0	0	0	0	0	0	0	0		
	20.9	0	0	0	0	0	0	0	0		
AC 1-5	30	15	46	82	5	5	18	61	87	16/18 = 88.88%	2/18 =11.12%

Categories	Rule Number	*one*								FCD %	
		Applied Rules	∑Satisfied Slice Sizes	Satisfied Weight	∑Violated Slice Sizes	Violated Weight	Function Frequency	Rule ∑ Function Slice Size	Rule Function Weight	FCSD%	FCVD%
Category 6	6.3	0	0	0	0	0	0	0	0	0/1 = 0%	1/1 = 100%
	11.3	0	0	0	0	0	0	0	0		
	12.1	0	0	0	0	0	0	0	0		
	17.4	0	0	0	0	0	0	0	0		
	19.1	0	0	0	0	0	0	0	0		
	19.2	0	0	0	0	0	0	0	0		
	19.7	-1	0	1	1	1	1	1	1		
FAC	37	16	46	83	6	6	19	62	88	16/19 = 84.21%	3/19 = 15.79%

***one* Function Calculation**

Appendix G: *main* Function Calculation Table

Categories	Rule Number	*main*								FCD %	
		Applied Rules	∑Satisfied Slice Sizes	Satisfied Weight	∑Violated SliceSizes	Violated Weight	Function Frequency	Rule ∑ Function Slice Size	Rule Function Weight	FCSD %	FCVD %
Category 1	4.1 + 7.1	+2/-3	2	4	3	9	5	5	25	12/15 = 80%	3/15 = 20%
	4.2	+2	2	4	0	0	2	2	4		
	5.1	+7	57	399	0	0	7	57	399		
	5.2	0	0	0	0	0	0	0	0		
	6.1	+1	4	4	0	0	1	4	4		
AC	5	4	64	364	15	45	15	68	432	12/15 = 80%	3/15 = 20%
Category 2	12.2	+2	2	4	0	0	2	2	4	19/20 = 95%	1/20 = 5%
	12.3	+2	25	50	0	0	2	25	50		
	13.4	+6	77	462	0	0	6	77	462		
	13.5	+6	77	462	0	0	6	77	462		
	13.6	-1	0	0	8	8	1	8	8		
	14.7	+1	53	53	0	0	1	53	53		
	17.1	0	0	0	0	0	0	0	0		
	17.5	+2	15	30	0	0	2	15	30		
AC 1-2	13	11	378	1836	23	62	35	325	1501	27/35 = 77.1%	8/35 =22.9%
Category 3	8.1	-1	0	0	53	53	1	53	53	11/14 = 78.57%	3/14 = 21.43%
	8.2	+1	53	53	0	0	1	53	53		
	8.3	0	0	0	0	0	0	0	0		
	8.6	+1	53	53	0	0	1	53	53		
	14.8	+2	27	54	0	0	2	27	54		
	16.1	+1/-1	53	53	1	1	2	54	108		
	16.2	+3	23	69	0	0	3	23	69		
	16.3	0	0	0	0	0	0	0	0		
	16.4	0	0	0	0	0	0	0	0		
	16.5	-1	0	0	53	53	1	53	53		
	16.8	+1	2	2	0	0	1	2	2		
	16.9	+2	8	16	0	0	2	8	16		
AC 1-3	25	20	219	2136	107	169	49	651	1962	38/49 = 77.55%	11/49= 22.45%
Category 4	8.12	+1	4	4	0	0	1	4	4	2/2 = 100%	0/2 = 0%
	9.2	+1	4	4	0	0	1	4	4		
AC 1-4	27	22	227	2144	107	169	51	659	1970	40/51 = 78.43%	11/51= 21.57%
Category 5	19.6	+2	2	4	0	0	2	2	4	4/4 = 100%	0/4 = 0%
	20.2	+1	1	1	0	0	1	1	1		
	20.9	+1	1	1	0	0	1	1	1		
AC 1-5	30	25	231	2150	107	169	55	663	1976	44/55 = 80%	11/55 = 20%

Categories	Rule Number	*main*								FCD%	
		Applied Rules	∑Satisfied Slice Sizes	Satisfied Weight	∑Violated SliceSizes	Violated Weight	Function Frequency	Rule ∑ Function Slice Size	Rule Function Weight	FCSD %	FCVD%
Category 6	6.3	-1	0	0	1	1	1	1	1	6/8 = 75%	2/8 = 25%
	11.3	0	0	0	0	0	0	0	0		
	12.1	+2	2	4	0	0	2	2	4		
	17.4	+2	2	4	0	0	2	2	4		
	19.1	+1	1	1	0	0	1	1	1		
	19.2	+1	1	1	0	0	1	1	1		
	19.7	-1	0	0	53	53	1	53	53		
FAC	37	31	237	2160	161	223	63	723	2040	50/63 = 79.37%	14/63= 20.63%

***main* Function Calculation**

Appendix I: SwapoAdd.c PCD Table

Category	PCD%	
	PCSD%	PCVD%
Category 1	24/30 = 80%	6/30 =20%
AC	24/30 = 80%	6/30 =20%
Category 2	21/25 = 84%	4/25 = 16%
AC	45/55 = 81.82%	10/55= 18.18%
Category 3	47/53 = 88.68%	6/53 = 11.32%
AC	92/108= 85.19%	16/108= 14.81%
Category 4	2/2 = 100%	0/2 = 0%
AC	94/110= 85.45%	16/110= 14.55%
Category 5	4/4 = 100%	0/4 = 0%
AC	98/114= 85.96%	17/114= 14.04%
Category 6	6/14 = 43%	8/14 = 57%
FAC	104/128= 81.25%	25/128 = 18.75%

SwapoAdd.c PCD Table

Appendix J: SwapoAdd.c Category Calculations Table

Categories	Rule Number	CATEGORY CALCULATIONS								
		Category Satisfied Frequency	Category Satisfied ∑Slice Sizes	Category Satisfied Weight	Category Violated Frequency	Category Violated Slice Sizes	Category Violated Weight	Category Frequency	Category ∑Slice Sizes	Rule Category Weight
Category 1	4.1 + 7.1	5	5	25	3	3	9	8	8	64
	4.2	5	5	25	0	0	0	5	5	25
	5.1	13	79	1027	0	0	0	13	79	1027
	5.2	0	0	0	3	13	39	3	13	39
	6.1	1	4	4	0	0	0	1	4	4
AC	5	24	93	1081	6	16	48	30	109	1159
Category 2	12.2	3	3	9	0	0	0	3	3	9
	12.3	2	25	50	0	0	0	2	25	50
	13.4	6	77	462	0	0	0	6	77	462
	13.5	6	77	462	0	0	0	6	77	462
	13.6	0	0	0	1	8	8	1	8	8
	14.7	2	59	118	2	17	34	4	76	304
	17.1	0	0	0	1	2	2	1	2	2
	17.5	2	15	30	0	0	0	2	15	30
AC 0-1	13	45	349	2212	10	41	44	55	392	2486
Category 3	8.1	3	23	69	1	53	35	4	76	304
	8.2	4	76	304	0	0	0	4	76	304
	8.3	5	19	95	0	0	0	5	19	95
	8.6	4	75	300	0	0	0	4	75	300
	14.8	2	27	54	0	0	0	2	27	52
	16.1	4	56	224	4	10	40	8	66	528
	16.2	9	63	567	0	0	0	9	63	567
	16.3	3	3	9	0	0	0	3	3	9
	16.4	3	3	9	0	0	0	3	3	9
	16.5	0	0	0	1	53	53	1	53	53
	16.8	2	4	8	0	0	0	2	4	8
	16.9	8	48	384	0	0	0	8	48	384
AC 0-2	25	92	746	4235	16	157	172	108	905	5099
Category 4	8.12	1	4	4	0	0	0	1	4	4
	9.2	1	4	4	0	0	0	1	4	4
AC 0-3	27	94	754	4243	16	157	172	110	913	5107

Categories	Rule Number	CATEGORY CALCULATIONS								
		Category Satisfied Frequency	Category Satisfied ∑Slice Size	Category Satisfied Weight	Category Violated Frequency	Category Violated ∑Slice Size	Category Violated Weight	Category Frequency	Category ∑Slice Size	Rule Category Weight
Category 5	19.6	2	2	4	0	0	0	1	4	4
	20.2	1	1	1	0	0	0	1	1	1
	20.9	1	1	1	0	0	0	1	1	1
AC 0-4	30	98	758	4249	16	157	172	113	919	5113
Category 6	6.3	0	0	0	1	1	1	1	1	1
	11.3	0	0	0	2	5	10	2	5	10
	12.1	2	2	4	0	0	0	2	2	4
	17.4	2	2	4	1	2	2	3	4	12
	19.1	1	1	1	0	0	0	1	1	1
	19.2	1	1	1	0	0	0	1	1	1
	19.7	0	0	0	4	56	224	4	56	224
FAC	37	104	764	4259	24	221	409	127	989	5366

SwapoAdd.c Category Calculations

Appendix K: Clause Table Equations

Clause Number	Slice Size	Clause Frequency	Clause Weight	Function Name	Applicable Rules	
					Satisfied	Violated
1	2.a	3	4	5	6	
					6.a	6.b

1. **Clause Number:** the Clause line number in the program code.
2. **Slice Size:** number of Clauses in a slice.
    a. **Clause ∑Slice Size:** is the total Slice Size for each program Clause that applied a MISRA rule which is equal to number of Clauses in a slice by slicing a Clause.
3. **Clause Frequency**: number of time a Clause is in a slice.
4. **Clause Weight:** is equal to Clause frequency multiply by Clause ∑Slice Size.
5. **Function Name:** is the function that the Clause belongs to.
6. **Applicable rules:** rules that applied in a program, function or category.
    a. **Satisfied Rules:** rules that applied and satisfied in a program, function or category.
    b. **Violated Rules:** rules that applied and violated in a program, function or category.

Appendix L: Robustness Grid Equations

In the following table, each number in the cell will be calculated by the equation that has the same number.

<table>
<tr><td rowspan="2">1</td><td rowspan="2">2</td><td colspan="8">3</td><td colspan="2">4</td><td colspan="2">5</td><td colspan="9">6</td></tr>
<tr><td>3.a</td><td>3.b</td><td>3.c</td><td>3.d</td><td>3.e</td><td>3.f</td><td>3.g</td><td>3.h</td><td rowspan="2">4.a</td><td rowspan="2">4.b</td><td rowspan="2">5.a</td><td rowspan="2">5.b</td><td rowspan="2">6.a</td><td rowspan="2">6.b</td><td rowspan="2">6.c</td><td rowspan="2">6.d</td><td rowspan="2">6.e</td><td rowspan="2">6.f</td><td rowspan="2">6.g</td><td rowspan="2">6.h</td><td rowspan="2">6.i</td></tr>
<tr><td>3.a (i, ii, iii, iv)</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="20">7</td></tr>
<tr><td colspan="10">8</td><td>9.a</td><td>9.b</td><td>10.a</td><td>10.b</td><td colspan="7">8</td><td>11</td></tr>
</table>

1. **Categories (Category 1 – Category 6):** set of rules share same characteristics, and ordered regarding to "*category selection conditions*" in Section 3.1.
2. **Rule Number:** MISRA C2 rule number, as shown in MISRA C2 document.
3. **Function Name:** each function and has these values:
    a. **Applied Rules:** MISRA C2 rules that are applied in one or more functions.
        i. **+n:** times of a rule been satisfied through a function = Function Satisfied Frequency.
        ii. **-n:** times of a rule been violated through a function = Function Violate Frequency
        iii. **0:** rule is not applicable in a function.

iv. Number of rules applied through a program in a category, which is equal to number of non-zero applied rules for a function. (NOT the sum of +n and –n, but how many times a rule whether +, –, or both been applied).

b. ∑**Satisfied Slice Size:** total size of all Clause slices that satisfy a rule in a function.

c. **Rule Satisfied Weight:** weight of satisfied rule which is equal to a function satisfied frequency for a rule multiply by ∑satisfied Slice Size for the same rule in a function for a category.

d. ∑**Violated Slice Size**: total size of all Clauses slices that violate a rule in a function.

e. **Rule Violated Weight:** weight of violated rule which is equal to a function satisfied frequency for a rule multiply by ∑violated Slice Size for the same rule in a function for a category.

f. **Rule Function Frequency:** times a rule been applied (+n +|-n|) in a function.

g. **Rule ∑ Function Slice Size:** ∑satisfied Slice Size + ∑violated Slice Size for a function.

h. **Rule Function Weight:** rule function weight multiply by rule ∑function Slice Sizes of a rule (which is equal to ∑satisfied Slice Size + ∑violated Slice Size).

4. FCD (Function Category Degree): has two values:

a. **FCSD (Function Category Satisfied Degree):** Robustness Degree for satisfied rules in a category for a function is equal to times of rules has been satisfied (∑+n) in same category for the same function

divided by all times a rule has been applied in the same category for the same function presented as a percentage.

b. **FCVD (Function Category Violated Degree):** Robustness Degree for violated rules in a category for a function is equal to times of rules has been violated (∑|-n|) in same category for the same function divided by all times a rule has been applied in the same category for the same function presented as a percentage.

5. **PCD (Program Category Degree):** Has two values:

   a. **PCSD (Program Category Satisfied Degree**): Robustness Degree for satisfied rules in a category for all functions is equal to times of rules has been satisfied (∑+n for all functions) in same category for all functions divided by all times a rule has been applied in the same category for all functions presented as a percentage.

   b. **PCVD (Program Category Violated Degree):** Robustness Degree for violated rules in a category for all functions is equal to times of rules has been violated (∑|-n| for all functions) in same category for all functions divided by all times a rule has been applied in the same category for all functions presented as a percentage.

6. **Category Calculations:**

*Frequency***:** times of a rule being applied through a program.

   a. **Category Satisfied Slice Size:** ∑Slice Size of a rule being satisfied through a category for all functions.

   b. **Category Satisfied Frequency:** times of a rule being satisfied through a category for all functions.

c. **Category Satisfied Weight:** sum of all satisfied rules weight in a category which is equal to sum of rules category satisfied frequency multiply by all rules ∑satisfied Slice Size in a category.

d. **Category Violated Slice Size:** ∑Slice Size of a rule being violated through a category for all functions.

e. **Category Violate Frequency:** times of a rule being violated through a category for all functions.

f. **Category Violated Weight:** sum of all violated rules weight in a category which is equal to sum of rules category violated frequency multiply by all rules ∑violated Slice Size in a category.

g. **Category ∑Slice Size:** total size of all Clauses slices that apply a rule in all functions.

h. **Category Frequency:** times of a rule being applied through a category for all functions. This means the rule frequency.

i. **Category Weight:** sum of all applicable rules weight in a category which is equal to sum of rules category frequency multiply by all rules ∑Slice Sizes (∑satisfied Slice Size +∑violated Slice Size) in a category.

7. **AC (Accumulative Categories):** Accumulative value for each column for all previous categories for a function.

8. **FAC (Function Accumulative Categories):** Accumulative value for each column for all categories for a function. The FAC, as with all other Robustness Degrees, has two values: FSAC is the Function Satisfaction Accumulative Degree and FVSC is the Function Violation Accumulative Degree.

9. **WCFD (Whole Categories Function Degree):** has two values:

    a. **WCFSD (Whole Categories Function Satisfied Degree):** ∑ all satisfied rules of all categories of a function divided by ∑ all applied rules in all categories of a function presented as a percentage. (In other words: crossing of FCSD column with FAC row).

    b. **WCFVD (Whole Categories Function Violated Degree):** ∑ all violated rules of all categories of a function divided by ∑ all applied rules of all categories of a function presented as a percentage. (In other words: crossing of FCVD column with FAC row).

10. **WPD (Whole Program Degree)** has two values:

    a. **WPSD (Whole Program Satisfied Degree):** ∑ all satisfied rules of a whole program divided by ∑ all applied rules of same whole program presented as a percentage. (In other words: crossing of PCSD column with FAC row).

    b. **WPVD Whole Program Violated Degree):** ∑ all violated rules of a whole program divided by ∑ all applied rules of same whole program presented as a percentage. (In other words: crossing of PCVD column with FAC row).

11. **WPW (Whole Program Weight):** ∑ all applied rules weight of whole program, which is equal to ∑ all applied rules frequency of whole program multiply by ∑ size of all applied rules slices of the whole program presented as a percentage. (In other words: crossing of Category Weight column with FAC row).

Appendix M: LDRA TBmisra test results (against MISRA C2)

```
 #include <stdio.h>
/* (M) STATIC VIOLATION     : 130 S : MISRA-C:2004 20.8,20.9,20.12: Included file
is not permitted. :      7F#include <stdio.h> */
   #define LAST 10
   #define ARRAY_SIZE 10
         typedef unsigned char x1;
         char array[ARRAY_SIZE] = "0123456789";
/* (O) STATIC VIOLATION     : 90 S : MISRA-C:2004 6.3: Basic type declaration
used. :     15Fchar */
/* (M) STATIC VIOLATION     : 404 S : MISRA-C:2004 9.2: Array initialisation has too
many items. :     16F  array [ 10 ] = "0123456789" ; */
/* (M) DATAFLOW VIOLATION   : 25 D : MISRA-C:2004 8.7: Scope of variable could
be reduced : array : 16 */
   void Incr(int *num, int i);
/* (O) STATIC VIOLATION     : 90 S : MISRA-C:2004 6.3: Basic type declaration
used. :     20F  int * num , */
/* (O) STATIC VIOLATION     : 90 S : MISRA-C:2004 6.3: Basic type declaration
used. :     21F  int i ) ; */
   void swap(int *a, int *b);
/* (O) STATIC VIOLATION     : 90 S : MISRA-C:2004 6.3: Basic type declaration
used. :     25F  int * a , */
/* (O) STATIC VIOLATION     : 90 S : MISRA-C:2004 6.3: Basic type declaration
used. :     26F  int * b ) ; */
   int one (int x);
/* (O) STATIC VIOLATION     : 90 S : MISRA-C:2004 6.3: Basic type declaration
used. :     28Fint */
/* (O) STATIC VIOLATION     : 90 S : MISRA-C:2004 6.3: Basic type declaration
used. :     30F  int x ) ; */
   int main()
/* (O) STATIC VIOLATION     : 90 S : MISRA-C:2004 6.3: Basic type declaration
used. :     32Fint */
   {
      int i, sum = 0, *a = 12,*b = 13, x = 3;
```

```
/* (O) STATIC VIOLATION     : 90 S : MISRA-C:2004 6.3: Basic type declaration
used. :     35F    int */
/* (O) XREF VIOLATION       : 49 X : MISRA-C:2004 5.7: Identifier reuse: var vs proc
param. : i :     36F      i , */
/*    See also line 6 SwapoAdd.c(SWAPOADD) */
/* (O) XREF VIOLATION       : 49 X : MISRA-C:2004 5.7: Identifier reuse: var vs proc
param. : a :     38F      * a = 12 , */
/*    See also line 7 SwapoAdd.c(SWAPOADD) */
/* (O) XREF VIOLATION       : 49 X : MISRA-C:2004 5.7: Identifier reuse: var vs proc
param. : b :     39F      * b = 13 , */
/*    See also line 7 SwapoAdd.c(SWAPOADD) */
/* (O) XREF VIOLATION       : 49 X : MISRA-C:2004 5.7: Identifier reuse: var vs proc
param. : x :     40F      x = 3 ; */
/*    See also line 8 SwapoAdd.c(SWAPOADD) */

       for ( i = 1; i <= LAST; i++ ) {
          Incr(&sum, i);
       }
       printf("sum = %d??/n", sum);
/* (M) STATIC VIOLATION     : 81 S : MISRA-C:2004 4.2: Use of trigraphs. :     54T
printf ( "sum = %d??/n" , sum ) ; */
       swap (&a,&b);
       int index;
/* (O) STATIC VIOLATION     : 90 S : MISRA-C:2004 6.3: Basic type declaration
used. :     59F      index ; */
         for (index = 0; index < ARRAY_SIZE; ++index)
         {
            one (x);
            printf("&array[index]=0x%p (array+index)=0x%p array[index]=0x%x\n",
/* (M) STATIC VIOLATION     : 87 S : Use of pointer arithmetic. :     72T        printf (
"&array[index]=0x%p (array+index)=0x%p array[index]=0x%x\n" , & array [ index ] , (
array + index ) , array [ index ] ) ; */
/* (M) STATIC VIOLATION     : 87 S : Use of pointer arithmetic. : 72 */
            &array[index], (array+index), array[index]);

         }
```

```
        return 0;
    }

    void Incr(int *num, int i) {
/* (O) STATIC VIOLATION     : 90 S : MISRA-C:2004 6.3: Basic type declaration
used. :     80F  int * num , */
/* (O) STATIC VIOLATION     : 90 S : MISRA-C:2004 6.3: Basic type declaration
used. :     81F  int i ) */
/* (O) XREF VIOLATION       : 49 X : MISRA-C:2004 5.7: Identifier reuse: var vs proc
param. : i : 81 */
/*    See also line 11 SwapoAdd.c(SWAPOADD) */
        *num = *num + i;
    }

    void swap(int *a, int *b)
/* (O) STATIC VIOLATION     : 90 S : MISRA-C:2004 6.3: Basic type declaration
used. :     88F  int * a , */
/* (O) STATIC VIOLATION     : 90 S : MISRA-C:2004 6.3: Basic type declaration
used. :     89F  int * b ) */
/* (O) XREF VIOLATION       : 49 X : MISRA-C:2004 5.7: Identifier reuse: var vs proc
param. : a : 88 */
/*    See also line 11 SwapoAdd.c(SWAPOADD) */
/* (O) XREF VIOLATION       : 49 X : MISRA-C:2004 5.7: Identifier reuse: var vs proc
param. : b : 89 */
/*    See also line 11 SwapoAdd.c(SWAPOADD) */
    {
        int temp= *a;
/* (O) STATIC VIOLATION     : 90 S : MISRA-C:2004 6.3: Basic type declaration
used. :     91F    int */
        *a= *b;
        *b= temp;
        printf ("pointer a is: %d\n",*a);
/* (M) DATAFLOW VIOLATION   : 45 D : MISRA-C:2004 21.1: Pointer not checked
for null before use : a :     95T    printf ( "pointer a is: %d\n" , * a ) ; */
/*    See also line 36 SwapoAdd.c(SWAPOADD) */
        printf ("pointer b is: %d\n",*b);
```

```
/* (M) DATAFLOW VIOLATION   : 45 D : MISRA-C:2004 21.1: Pointer not checked
for null before use : b :     96T    printf ( "pointer b is: %d\n" , * b ) ; */
/*    See also line 37 SwapoAdd.c(SWAPOADD) */
   }

   int one (int x)
/* (O) STATIC VIOLATION     : 90 S : MISRA-C:2004 6.3: Basic type declaration
used. :     99Fint */
/* (O) STATIC VIOLATION     : 90 S : MISRA-C:2004 6.3: Basic type declaration
used. :    101F  int x ) */
/* (O) XREF VIOLATION       : 49 X : MISRA-C:2004 5.7: Identifier reuse: var vs proc
param. : x : 101 */
/*    See also line 11 SwapoAdd.c(SWAPOADD) */
   {
         printf ("%d\n", x);
         return 1;
   }
```

Appendix N: FlexeLint test results (against MISRA C2)

```
FlexeLint for C/C++ (Unix/386) Vers. 9.00i, Copyright Gimpel Software 1985-2012
--- Module:   SwapoAdd.c (C)
        char array[ARRAY_SIZE] = "0123456789";
       ^
SwapoAdd.c:5:8: Note 970: Use of modifier or type 'char' outside of a typedef [MISRA 2004
Rule 6.3, advisory]
                                    ^
SwapoAdd.c:5:45: Info 784: Nul character truncated from string
   void Incr(int *num, int i);
            ^
SwapoAdd.c:6:14: Note 970: Use of modifier or type 'int' outside of a typedef [MISRA 2004
Rule 6.3, advisory]
                   ^
SwapoAdd.c:6:24: Note 970: Use of modifier or type 'int' outside of a typedef [MISRA 2004
Rule 6.3, advisory]
   void swap(int *a, int *b);
            ^
SwapoAdd.c:7:14: Note 970: Use of modifier or type 'int' outside of a typedef [MISRA 2004
Rule 6.3, advisory]
                   ^
SwapoAdd.c:7:22: Note 970: Use of modifier or type 'int' outside of a typedef [MISRA 2004
Rule 6.3, advisory]
   int one (int x);
   ^
SwapoAdd.c:8:4: Note 970: Use of modifier or type 'int' outside of a typedef [MISRA 2004
Rule 6.3, advisory]
            ^
SwapoAdd.c:8:13: Note 970: Use of modifier or type 'int' outside of a typedef [MISRA 2004
Rule 6.3, advisory]
   int main()    {
   ^
SwapoAdd.c:9:4: Note 970: Use of modifier or type 'int' outside of a typedef [MISRA 2004
Rule 6.3, advisory]
       int i, sum = 0, *a = 12,*b = 13, x = 0;
```

```
    ^
SwapoAdd.c:10:8: Note 970: Use of modifier or type 'int' outside of a typedef [MISRA 2004
Rule 6.3, advisory]
                    ^
SwapoAdd.c:10:31: Error 64: Type mismatch (initialization) (int * = int) [MISRA 2004 Rule
1.2, required], [MISRA 2004 Rule 8.4, required]
                        ^
SwapoAdd.c:10:39: Error 64: Type mismatch (initialization) (int * = int) [MISRA 2004 Rule
1.2, required], [MISRA 2004 Rule 8.4, required]
        Incr(&sum, i);      }
             ^
SwapoAdd.c:13:21: Note 934: Taking address of near auto variable 'sum' of type 'int' (arg.
no. 1) [MISRA 2004 Rule 1.2, required]
    printf("sum = %d ??/n", sum);
        ^
SwapoAdd.c:14:15: Warning 584: Trigraph sequence (??/) detected [MISRA 2004 Rule 4.2,
required]
                        ^
SwapoAdd.c:14:36: Warning 534: Ignoring return value of function 'printf(const char *, ...)' of
type 'int (const char *, ...)' (compare with line 1) [Encompasses MISRA 2004 Rule 16.10,
required]
SwapoAdd.c:1:0: Info 830: Location cited in prior message
    swap (&a,&b);
        ^
SwapoAdd.c:15:16: Note 918: Prototype coercion (arg. no. 1) of pointers [MISRA 2004 Rule
10.2, required]
SwapoAdd.c:15:16: Error 64: Type mismatch (arg. no. 1) (int * = int **) [MISRA 2004 Rule
1.2, required], [MISRA 2004 Rule 8.4, required]
SwapoAdd.c:15:16: Note 934: Taking address of near auto variable 'a' of type 'int *' (arg. no.
1) [MISRA 2004 Rule 1.2, required]
             ^
SwapoAdd.c:15:19: Note 918: Prototype coercion (arg. no. 2) of pointers [MISRA 2004 Rule
10.2, required]
SwapoAdd.c:15:19: Error 64: Type mismatch (arg. no. 2) (int * = int **) [MISRA 2004 Rule
1.2, required], [MISRA 2004 Rule 8.4, required]
SwapoAdd.c:15:19: Note 934: Taking address of near auto variable 'b' of type 'int *' (arg. no.
```

2) [MISRA 2004 Rule 1.2, required]

```
    int index;
    ^
```

SwapoAdd.c:16:8: Error 42: Expected a statement

```
          ^
```

SwapoAdd.c:16:17: Note 960: Violates MISRA 2004 Required Rule 14.3, null statement not in line by itself

```
      for (index = 0; index < ARRAY_SIZE; ++index)        {
          ^
```

SwapoAdd.c:17:16: Error 40: Undeclared identifier 'index' [MISRA 2004 Rule 1.2, required]

```
                ^
```

SwapoAdd.c:17:25: Error 63: Expected an lvalue

```
                 ^
```

SwapoAdd.c:17:27: Error 40: Undeclared identifier 'index' [MISRA 2004 Rule 1.2, required]

```
                                  ^
```

SwapoAdd.c:17:49: Error 40: Undeclared identifier 'index' [MISRA 2004 Rule 1.2, required]
SwapoAdd.c:17:49: Error 52: Expected an lvalue
SwapoAdd.c:17:49: Note 961: Violates MISRA 2004 Advisory Rule 12.13, increment or decrement combined with another operator

```
      one (x);
                ^
```

SwapoAdd.c:18:26: Warning 534: Ignoring return value of function 'one(int)' of type 'int (int)' (compare with line 8) [Encompasses MISRA 2004 Rule 16.10, required]
SwapoAdd.c:8:0: Info 830: Location cited in prior message

```
          &array[index], (array+index), array[index]);     }
              ^
```

SwapoAdd.c:20:22: Error 40: Undeclared identifier 'index' [MISRA 2004 Rule 1.2, required]

```
                       ^
```

SwapoAdd.c:20:37: Error 40: Undeclared identifier 'index' [MISRA 2004 Rule 1.2, required]

```
                            ^
```

SwapoAdd.c:20:43: Warning 626: argument no. 3 inconsistent with format

```
                                  ^
```

SwapoAdd.c:20:51: Error 40: Undeclared identifier 'index' [MISRA 2004 Rule 1.2, required]

```
                                     ^
```

SwapoAdd.c:20:58: Warning 534: Ignoring return value of function 'printf(const char *, ...)' of type 'int (const char *, ...)' (compare with line 1) [Encompasses MISRA 2004 Rule 16.10,

required]
SwapoAdd.c:1:0: Info 830: Location cited in prior message
   void Incr(int *num, int i) {
            ^
SwapoAdd.c:23:14: Note 970: Use of modifier or type 'int' outside of a typedef [MISRA 2004 Rule 6.3, advisory]
                  ^
SwapoAdd.c:23:24: Note 970: Use of modifier or type 'int' outside of a typedef [MISRA 2004 Rule 6.3, advisory]
   void swap(int *a, int *b)   {
            ^
SwapoAdd.c:26:14: Note 970: Use of modifier or type 'int' outside of a typedef [MISRA 2004 Rule 6.3, advisory]
                 ^
SwapoAdd.c:26:22: Note 970: Use of modifier or type 'int' outside of a typedef [MISRA 2004 Rule 6.3, advisory]
      int temp= *a;
      ^
SwapoAdd.c:27:8: Note 970: Use of modifier or type 'int' outside of a typedef [MISRA 2004 Rule 6.3, advisory]
      printf ("pointer a is: %d\n",*a);
                                ^
SwapoAdd.c:30:40: Warning 534: Ignoring return value of function 'printf(const char *, ...)' of type 'int (const char *, ...)' (compare with line 1) [Encompasses MISRA 2004 Rule 16.10, required]
SwapoAdd.c:1:0: Info 830: Location cited in prior message
      printf ("pointer b is: %d\n",*b);   }
                                ^
SwapoAdd.c:31:40: Warning 534: Ignoring return value of function 'printf(const char *, ...)' of type 'int (const char *, ...)' (compare with line 1) [Encompasses MISRA 2004 Rule 16.10, required]
SwapoAdd.c:1:0: Info 830: Location cited in prior message
   int one (int x)   {
   ^
SwapoAdd.c:33:4: Note 970: Use of modifier or type 'int' outside of a typedef [MISRA 2004 Rule 6.3, advisory]

```
          ^
SwapoAdd.c:33:13: Note 970: Use of modifier or type 'int' outside of a typedef [MISRA 2004
Rule 6.3, advisory]
        printf ("%d\n", x);
                        ^
SwapoAdd.c:34:26: Warning 534: Ignoring return value of function 'printf(const char *, ...)' of
type 'int (const char *, ...)' (compare with line 1) [Encompasses MISRA 2004 Rule 16.10,
required]
SwapoAdd.c:1:0: Info 830: Location cited in prior message
    --- Wrap-up for Module: SwapoAdd.c
SwapoAdd.c:4:0: Info 751: local typedef 'x1' of type 'unsigned char' (line 4, file SwapoAdd.c)
not referenced
:0:0: Note 960: Violates MISRA 2004 Required Rule 8.7, could define variable at block
scope: array
--- Global Wrap-up
SwapoAdd.c:5:0: Warning 552: Symbol 'array' of type 'char [10]' (line 5, file SwapoAdd.c) not
accessed
SwapoAdd.c:5:0: Info 843: Variable 'array' of type 'char [10]' (line 5, file SwapoAdd.c) could
be declared as const
SwapoAdd.c:1:0: Warning 526: Symbol 'printf(const char *, ...)' of type 'int (const char *, ...)'
(line 1, file SwapoAdd.c) not defined
:0:0: Note 974: Worst case function for stack usage: 'main' is finite, requires 80 bytes total
stack in calling 'swap'. See +stack for a full report. [MISRA 2004 Rule 16.2, required]
:0:0: Note 900: Successful completion, 57 messages produced
```

Appendix O: LDRA TenDRA test results (against ISO 1990)

"*SwapoAdd.c*", line 1 column 2: Warning:
[ISO 6.8]: Indented preprocessing directive.

"*SwapoAdd.c*", line 2 column 5: Warning:
[ISO 6.8]: Indented preprocessing directive.

"*SwapoAdd.c*", line 3 column 5: Warning:
[ISO 6.8]: Indented preprocessing directive.

"*SwapoAdd.c*", line 11 column 32: Error:
[ISO 6.5.7]: In initialization of 'a'.
[ISO 6.3.4]: Conversion of nonzero value of type 'int' to type 'int *'.
[ISO 6.3.16]: Can't perform this conversion by assignment.
[ISO 6.5.7]: Initializers are converted as if by assignment.

"*SwapoAdd.c*", line 11 column 40: Error:
[ISO 6.5.7]: In initialization of 'b'.
[ISO 6.3.4]: Conversion of nonzero value of type 'int' to type 'int *'.
[ISO 6.3.16]: Can't perform this conversion by assignment.
[ISO 6.5.7]: Initializers are converted as if by assignment.

"*SwapoAdd.c*", line 16 column 36: Warning:
[ISO 6.3.2.2]: In call of function 'printf'.
[ISO 6.6.3]: Discarded function return.

"*SwapoAdd.c*", line 17 column 20: Error:
[ISO 6.3.2.2]: In call of function 'swap'.
[ISO 6.1.2.6]: The types 'int *' and 'int' are incompatible.
[ISO 6.3.4]: Types in pointer conversion should be compatible.
[ISO 6.3.16]: Can't perform this conversion by assignment.
[ISO 6.3.2.2]: Argument 1 is converted to parameter type.

"*SwapoAdd.c*", line 17 column 20: Error:
[ISO 6.3.2.2]: In call of function 'swap'.
[ISO 6.1.2.6]: The types 'int *' and 'int' are incompatible.
[ISO 6.3.4]: Types in pointer conversion should be compatible.
[ISO 6.3.16]: Can't perform this conversion by assignment.
[ISO 6.3.2.2]: Argument 2 is converted to parameter type.

"*SwapoAdd.c*", line 18 column 17: Error:
[ISO 6.6.2]: Declaration statement should be at start of block.

"*SwapoAdd.c*", line 21 column 21: Warning:
[ISO 6.3.2.2]: In call of function 'one'.
[ISO 6.6.3]: Discarded function return.

"*SwapoAdd.c*", line 23 column 59: Warning:
[ISO 6.3.2.2]: In call of function 'printf'.
[ISO 6.6.3]: Discarded function return.

"*SwapoAdd.c*", line 38 column 41: Warning:
[ISO 6.3.2.2]: In call of function 'printf'.
[ISO 6.6.3]: Discarded function return.

"*SwapoAdd.c*", line 39 column 41: Warning:
[ISO 6.3.2.2]: In call of function 'printf'.
[ISO 6.6.3]: Discarded function return.

"*SwapoAdd.c*", line 44 column 25: Warning:
[ISO 6.3.2.2]: In call of function 'printf'.
[ISO 6.6.3]: Discarded function return.

Appendix P: Variance.c program

```
1-  #include<stdio.h>
2-  #define MAX 1024
3-  main()
4-  {       float x[MAX];
5-      float var1,var2,var3,var4,var5;
6-      float ssq, avg, dev;
7-      float t1,t2,t3;
8-      int ii,jj,n;
9-      t1=0.0;
10-     t2=0.0;
11-     t3=0.0;
12-     ssq=0.0;
13-     scanf("%d", &n);
14-     for (ii=0; ii<n; ii=ii +1)
15-     {   scanf("%f", &x[ii]);
16-         t1 = t1+x[ii];
17-         ssq =ssq +x[ii]*x[ii];}
18-     avg =t1/n;
19-     var3=(ssq - n*avg*avg)/ (n-1);
20-     var4=(ssq - t1*avg)/ (n-1);
21-     t1=t1*t1/n;
22-     var2=(ssq-t1)/(n-1);
23-     for (jj=0; jj<n; jj=jj+1)
24-     {   dev = x[jj]-avg;
25-         t2=t2+dev;
26-         t3=t3+dev*dev;}
27-     var1=t3/(n-1);
28-     var5= (t3-t2*t2/n)/(n-1);
29-     printf("variance 1 (two pass):%f\n", var1);
30-     printf("variance 2 (one pass, using square of sum):%f\n",var2);
31-     printf("variance 3 (one pass, using average):%f\n",var3);
32-     printf("variance 4 (one pass, using average, sum):%f\n",var4);
33-     printf("variance 5 (two pass, corrected):%f\n",var2);}
```

Appendix Q: n_char program

```
#include <stdio.h>[1]
#include <string.h>[2]
void main()[3]
{
    int  n;[4]
    char string[50];[5]
    strcpy[6](string[7], "Hello World"[8]);
    n = n_char[9](string[10]);
    printf[11]("Length of string = %d\n"[12], n[13]);
}

int n_char[14](char string[][15])
{
    int n;[16]
    n = strlen[17](string[18]);
    if (n > 50) [19]
      printf[20]("String is longer than 50 characters\n"[21]);
    return n;[22]
}
```