Journal of Information Assurance and Security.
ISSN 1554-1010 Volume 14 (2019) pp. 011-019
© MIR Labs, www.mirlabs.net/jias/index.html

An Empirical Validation of Functional Redundancy Semantic Metric as Error Detection Indicator

Dalila Amara[1], Ezzeddine Fatnassi[1] and Latifa Rabai[1,2]

[1] Higher Institute of Management of Tunis, University of Tunis,
SMART Lab, Tunis, Tunisia,
dalila.amara@gmail.com, Ezzeddine.fatnassi@gmail.com

[2] College of Business, University of Buraimi, Al Buraimi,
P.C. 512, Sultanate of Oman
Latifa.rabai@gmail.com

***Abstract*: Software metrics are widely discussed and used as quantitative software quality measures. The main objective of these metrics is to help testers, managers and developers monitor the degree of quality of their systems. They are generally classified into syntactic and semantic categories. Most proposed software metrics are successfully used in measuring internal quality attributes like complexity and coupling. Concerning external attributes like reliability, a suite of four sematic metrics is proposed to assess programs' redundancy in order to reflect their ability to tolerate faults and monitor their reliability. Literature shows that the key limitation of the different metrics composing this suite is the lack of empirical studies to validate them. Consequently, the main purpose of this study is to empirically validate one of these metrics namely functional redundancy, as measure of program function redundancy in one side and as an error detection indicator in the other side. Results show that this metric is strongly correlated with the program output redundancy. This indicates that the functional redundancy metric is useful as a measure of program redundancy. Moreover, we show that it is useful as an error detection indicator.**

Keywords: Software Dependability, Fault Tolerance, Software Redundancy, Semantic Metrics, Functional Redundancy Metric.

I. Introduction

Software quality is defined by the ISO/IEC standard [1] as the capability of software product to satisfy stated and implied needs when used under specified conditions. It is generally represented through a number of characteristics or attributes that may be internal i.e complexity or external like reliability [2, 39]. These attributes need to be quantitatively measured in order to help developers, testers and managers estimate or predict the degree of their systems' quality during the different phases of the development life cycle.

Software quality measurement is the process of collecting information related to the presented quality attributes [3]. The common discussed way to perform software quality measurement is software metrics [2, 3, 38]. They are defined as quantitative measures of the different quality characteristics [4, 5].

Numerous software metrics are proposed in the literature. They are successfully used in measuring various quality attributes [4, 6, 7] and they are generally classified into two categories which are syntactic and semantic metrics. Syntactic metrics are proposed to measure the different attributes related to the program structure like complexity and cohesion.

The well-defined syntactic metrics include those of Chidamber and Kemerer [6], Li and Henry [8] and Abreu et al. [9] which are proposed to measure the Object- Oriented (OO) properties i.e complexity, cohesion, inheritance, etc. Concerning sematic metrics, they are proposed to measure different quality attributes including the cited ones based on the program functionality. Examples of semantic metrics suites include those proposed by Etzkorn and Delugash [4], Marcus and Poshyvanyk [10] and Stein et al. [11]. Further details about software metrics are presented in [7, 12, 36, 37, 40].

As noted above, most of the proposed syntactic and semantic metrics are used to assess internal quality attributes i.e complexity and cohesion [2, 40]. For the external ones, i.e reliability, few studies focused on their assessment. Among them, Mili et al. [13] proposed a suite of four semantic metrics whose objective is to assess programs information redundancy. The measured redundancy is then used to reflect the ability of these programs to tolerate faults through error detection, masking and recovery. One of these metrics namely functional redundancy, is proposed to help detect the program errors based on information redundancy assessment. Information redundancy is one of software redundancy types and means that use extra information than it is needed [5, 14].

Despite the importance of this suite, literature shows that it is theoretically presented and manually computed. Also, there is a lack of empirical studies to validate it. The metrics validation consists of demonstrating the usefulness of the metric to measure what it is purported to measure based on case studies and experiments [4, 10, 11].

Consequently, we proposed in our previous works an automated way to automatically generate the different metrics composing this suite [15, 16]. The performed studies show that it is possible to automatically generate these metrics for different Java programs. However, further studies still required to demonstrate that these metrics are measures of programs' redundancy in one side and that they are useful as error detection and masking indicators in the other side.

Consequently, we focus in this paper on one of these metrics namely functional redundancy semantic metric and the main objectives are as follows:

- Demonstrate that the functional redundancy metric is useful to quantitatively assess the redundancy of the program function.
- Demonstrate that the functional redundancy metric is also useful as an error detection indicator.

The remainder of this paper is organized as follows. In section 2, we present the redundancy use to perform fault tolerant software systems. In section 3, we present the well-defined software metrics for quality assessment including those proposed for fault tolerance assessment. Section 4 describes the functional redundancy semantic metric, its purpose, the mathematical formulation as well as an illustrative example. Section 5 presents the validation methodology of this metric and describes the different case studies. Data analysis and discussion are presented in section 6. Conclusion and perspectives will be presented in section 7.

II. Software Redundancy Use for Fault Tolerance

In this section, we present the key concepts on which this study is constructed. These concepts are fault tolerance and fault tolerance techniques, software redundancy and software redundancy applications. The objective is to identify the relationship between these concepts and to clarify the motivation of our study.

A. *Fault tolerance techniques*

Software fault tolerance is one of the important means that help to achieve dependable software systems as shown in Figure. 1.

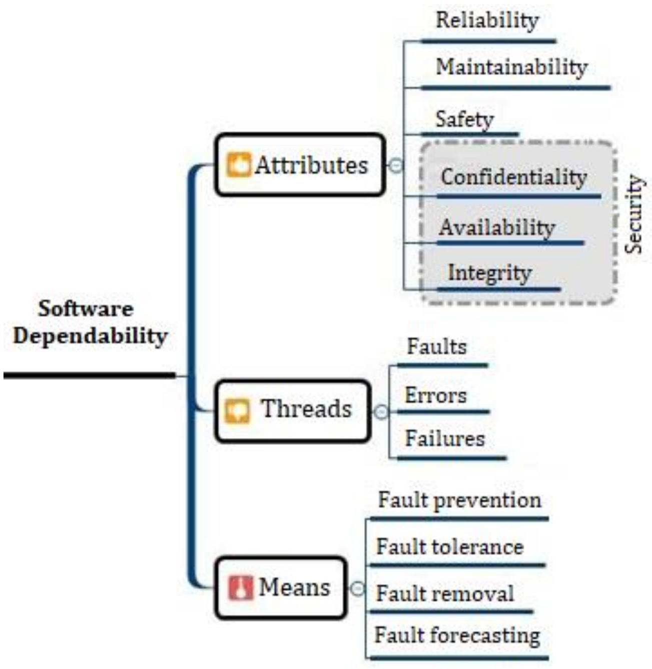


Figure 1. Software dependability concepts

Figure.1. shows that software dependability is described through different quality attributes including reliability, maintainability, safety and security. The main purpose is to avoid software threads (faults, errors and failures). This may be achieved using different dependability means which are fault prevention, fault removal, fault forecasting and fault tolerance.

Fault prevention consists in limiting the introduction of faults during software construction using formal methods [17, 18]. Fault removal aims to reduce the number and severity of existing faults by detecting and eliminating them using software testing and formal inspection [3, 19, 20]. Fault forecasting techniques aim to estimate the present number of faults in the system, their future occurrence as well as their consequences [3, 21].

Concerning fault tolerance, it aims to provide the required mechanisms to sidestep the manifestation of the remaining faults which avoid the system failure [20, 21]. Different studies show up the importance of fault tolerance against the other ones for two main reasons. First, fault avoidance and removal are based on exhaustive testing and program correctness and there are no reliable tools to guarantee that complicated software systems are error-free [3]. Second, it is not possible to remove all faults [22].

Consequently, software fault tolerance is proposed as an alternate technique since it allows the presence of faults in the system, but at the same time, it provides the required mechanisms to guarantee the software operation and delivery through failure avoidance [23]. It is based on three major phases [24, 25]:

- Error detection: is the ease of detecting errors in the state of a program in execution [24]. Two redundancy semantic metrics are proposed for error detection namely state redundancy and functional redundancy [3].
- Error recovery: identifies the erroneous state before its substitution with an error-free one [25]. Mili et al. [13] proposed a semantic metric for error recovery termed error non-determinacy.

- Error compensation: provides fault masking. A semantic metric called error non-injectivity as a measure of the program' ability to mask errors is proposed [13].

Achieving fault tolerance systems requires the use of different techniques in the different presented phases. So, numerous fault tolerance techniques are proposed in the literature.

Different criteria are used to classify these techniques like program versioning [23], diversity [21], the adjudication and execution scheme [20]. So, we present in Figure.2 most of these techniques:

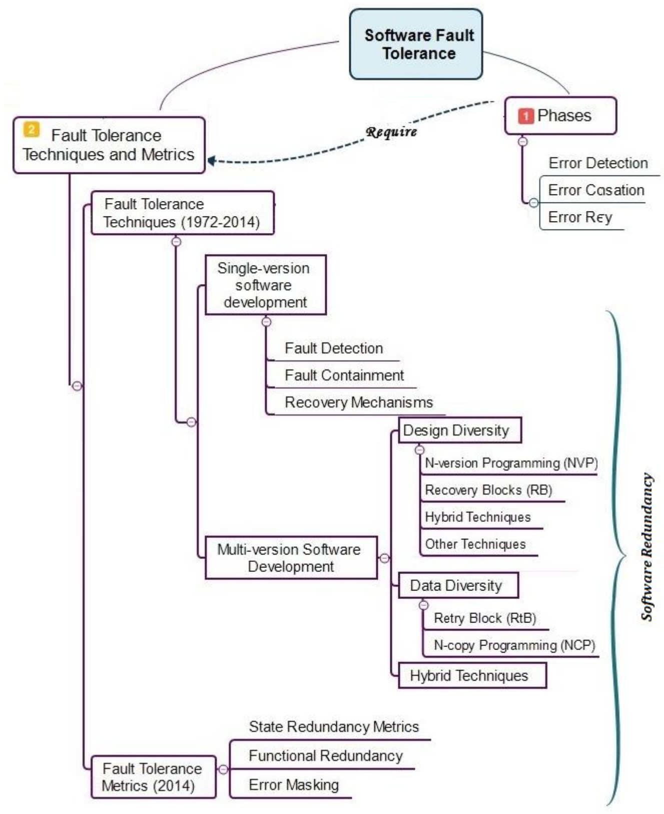


Figure 2. Fault tolerance techniques and metrics

Figure.2. shows that fault tolerance techniques are generally classified into two major categories which are single version and multi version techniques. Single version techniques focus on improving the fault tolerance of a single piece of software [23]. Multi version programming techniques focus on improving the fault tolerance of different versions of a program having the same specification [21]. They consist of three major groups which are design diversity, data diversity and hybrid techniques that combine the two previous ones (See Figure.2.). For further details about these techniques, their advantages and disadvantages, readers are advised to see Pullum [20] and Rizwan [22].

As presented in Figure. 2., most of these techniques are based around a common concept that is software redundancy that will be described in the following subsection.

B. *Software redundancy*

Redundancy is the basic concept of fault tolerance techniques and was firstly used in hardware systems by providing more physical copies of components like redundant processors, memories, buses, or power supplies in order to improve the reliability of basic components [21, 39].

- In software systems, software redundancy is defined as the duplication of state information or system function [14, 26]. For Dubrova [21], it is useful in detecting and correcting faults. In fact, software redundancy has different forms [26, 27]:
- Information redundancy: is related to the coding theory originally defined by Richard Hamming [33] and Claude Shannon [34]. It indicates the additional used information (bits) to represent a program state than it is required. Examples include parity code bit, error correcting codes, Hamming code and check sums [21].
- Functional redundancy: consists of using the same program specification to generate different algorithms or programs versions which perform the same functionality [27].
- Time redundancy: consists on repeating the execution of the failed process. For Pullum [20] and Dubrova [21], this type of software redundancy is not likely appropriate to be applied in real time applications and the other redundancy techniques are more appropriate for these applications.

C. *Redundancy applications*

As mentioned above, most of fault tolerance techniques are based on software redundancy. Literature shows that it is exploited for different applications [7]. Following are the most discussed ones:

- In 1976, redundancy is exploited through N-version programming technique and results show that this approach is efficient to achieve fault tolerant systems [19]. This technique is also used to compare the probability of failure between single and N-version systems.
- In 1990, redundancy is exploited for self-checking programs [24]. It is useful to check the own program' behavior during execution [28].
- In 1991, redundant software systems configured in a N-version structure are used to improve reliability [29].
- In 2003, redundancy structured in mutation testing is exploited through the injection of faults in numerous program versions. Results show its aptness to reflect systems' ability to tolerate faults [23].
- In 2009, redundancy is exploited to identify the code fragments having the same functionality based on the code semantic analysis. Results show the existence of functionally equivalent fragments having different syntax which leads to optimize the code and facilitate its understanding [30].
- In 2014, redundancy is exploited to identify the relationship between structural similarity, vocabulary similarity and method name similarity. Results show that the source code similarity cannot be always be reflected through these three concepts [31].
- In 2014, redundancy is exploited to reflect the ability of software systems to tolerate faults [13].
- In 2015, redundancy is also exploited to identify code fragments semantic similarity. Results show that it is possible to discriminate the minimally different code from

the truly redundant one in one side, and the low-level code redundancy from high-level algorithmic redundancy in the other side.

- In 2018, Asghari et al. [27] proposed a method to detect transient faults in the processor core in order to improve real time multitask system' fault tolerance based on information redundancy.

To sum up, redundancy is useful in different applications as presented above. However, literature shows that there is a lack of studies focusing on its assessment [5]. So, as software metrics are widely discussed as quantitative measures of software quality attributes like complexity and cohesion [4], Mili et al. [13] proposed a suite of four semantic metrics to assess programs' redundancy as it is shown in Figure.2. The objective is to reflect the ability of programs to tolerate faults and monitor their reliability.

The key limitation of this suite is that is manually computed and theoretically presented. So, further experimental studies still required to demonstrate if the proposed metrics are useful as measures of program redundancy and how this redundancy may be exploited to reflect the program ability to tolerate faults [13]. The different metrics composing this suite are described in the following section.

III. Software Metrics for Fault Tolerance Assessment

This section presents the use of software metrics for fault tolerance assessment. So, we start by presenting a summarize of the well-defined syntactic and semantic suites as measures of various quality attributes. Then, we detail the proposed semantic metrics suite of Mili et al. [13] as redundancy measures.

A. Software metrics for quality assessment

Numerous syntactic and semantic metrics are proposed in literature to assess different quality attributes.

Most of the proposed syntactic metrics are used to assess internal quality attributes. For instance, we can cite those proposed by Chidamber and Kemerer (C&K) [6], Li and Henry [8], Abreu et al. [9], Bansiya and Davis [32] and much others [36, 37]. These suites are proposed as measures of programs complexity, cohesion, inheritance, coupling, and much other OO attributes related to the program structure [2, 39].

The proposed semantic metrics suites include those of Etzkorn and Delugash [4], Stein et al. [11], Cox et al. [35], Marcus and Poshyvanyk [10] and Mili et al. [13]. Compared to syntactic metrics, which are related to the program structure (attributes, number of methods, etc), semantic metrics depend on the program understanding through comments, identifier, concepts and concepts relationships.

Most of the presented software metrics are used to assess internal attributes. However, the suite proposed by Mili et al. [13] aims to reflect the program ability to tolerate faults based on redundancy assessment. So, we present in the following subsection, the different metrics composing this suite.

B. Redundancy-based semantic metrics suite

Four basic semantic metrics are proposed by Mili et al. [13] to assess program redundancy. These metrics are namely state redundancy, functional redundancy, non-injectivity and non-determinacy:

- State redundancy: is proposed to quantify the program information redundancy expressed in Shannon bits. It measures the excess of information in both initial and final program states. The main purpose of this metric is to help detect errors as one of the main fault tolerance phases [3]. For more details about the automated computing of this metric, readers are invited to see our previous work [15].
- Functional redundancy: is proposed to quantify the program redundancy that is the excess data of the output generated by the program function. The purpose of this metric is to help detect errors in one side, and to check the correctness of the program function in another side [13, 14]. For more details about the automated computing of this metric, readers are invited to see our previous work [16].
- Non-injectivity: is proposed to assess program redundancy using the non-injectivity of this program. The non-injectivity expresses the amount (bits) of uncertainty do we have about the initial state if we know the final state. The purpose of this metric is to mask errors by producing a subsequent state that bears no trace of the error.
- Non-determinacy: The non-determinacy metric (ND) reflects the program' specification flexibility. It means the amount of information the program may use to tolerate faults without violating its specification. Thus, a program is determinant if it doesn't tolerate faults; the final state of the program is always different from the expected one. A program is said non-determinant (has a non-deterministic specification) when it fails to compute its exact intended function and still satisfying the specification, so the final state may be equal to the expected one. Consequently, the non-determinacy metric is proposed to measure the extent to which a program may deviate from its intended behavior without violating its specification [13].

We recall that we focus in this study on the empirical assessment of the functional redundancy semantic metric.

IV. Functional Redundancy Semantic Metric

We describe in this section the main purpose of the functional redundancy semantic metric, its mathematical formulation as well as an illustrative example to clarify its purpose.

The functional redundancy metric reflects the excess output data generated by a program function in order to verify the proper execution of the function [14].

Mathematically speaking, this metric is equated with the non-surjectivity of the program function which means that not all program' outputs (final states) are mapped to at least one input (initial state) [13]. Hence, the functional redundancy of a program g on a space S is denoted by φ (g) and defined by:

$\varphi(g) = (H(S) - H(Y))/H(Y)$ (1)

- H(S) is the state space of the program: the maximum value (size in bits) that the declared program variables may take,

- Y is the random variable (range or output space) of g defined as the set of final states of the program's function,
- H(Y) = H(σf), is the entropy of the output produced by g defined as the number of bits required to store the result of the program' execution,
- φ(g) is the functional redundancy of the program g.

From the value of this metric, three possible interpretations are drawn [13, 14].

- if φ(g) = 0: there is no scope for checking any property since all bits are used, the input and output spaces are equal.
- if 0 < φ(g) < 1: we can check part of the result against redundant information,
- if φ(g) > 0: there are larger bits of redundancy when executing the program.

As an illustrative example, consider two functions F1=x and F2=x%4, and B5, the input range that is a set of natural numbers represented in a word of five bits as shown in Figure.3.

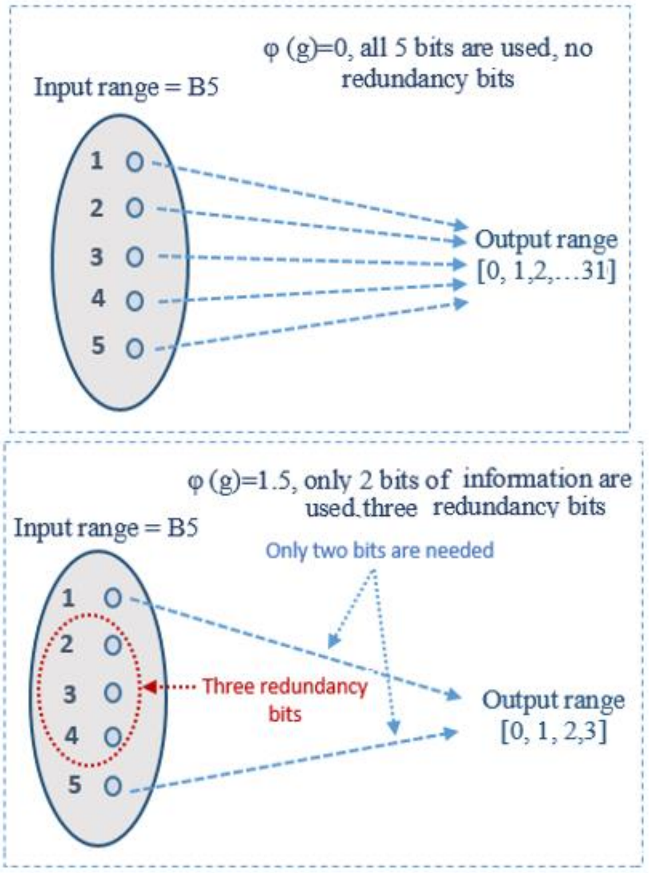

φ (g)=0, all 5 bits are used, no redundancy bits
Input range = B5
1
2
3
4
5
Output range [0, 1,2,…31]
φ (g)=1.5, only 2 bits of information are used three redundancy bits
Input range = B5
Only two bits are needed
Three redundancy bits
Output range [0, 1, 2,3]

Figure 3. Illustrative Example of functional redundancy metric

Figure.3 shows that for the first function, the possible output range is [0…31], hence all five bits are used and then there is no redundancy. However, for the second function, the output range is [0, 1, 2, 3]. In other words, only two bits of information are needed, whereas the three other ones are redundancy bits [14].

V. Empirical Validation of Functional redundancy metric

This section presents the validation methodology, the data collection and the computing process of the functional redundancy metric.

A. *Validation methodology*

As presented in the previous sections, the functional redundancy semantic metric (FR) is proposed to assess programs redundancy in order to help detecting errors and to check the correctness of the program function output [13].

To validate this hypothesis, we are based on the following research questions (RQ):

- ***RQ1:*** *Does the functional redundancy metric is useful to measure programs redundancy?*
- ***RQ2:*** *Does the functional redundancy semantic metric could be used to detect errors?*

B. *Data collection*

In order to answer the previous RQs, we will perform two main experiments. For each experiment, we resort to Java language and the Eclipse development environment (version: Neon.3 Release (4.6.3)) to automatically generate the presented functional redundancy metric.

1) Experiment 1: Functional redundancy metric for redundancy assessment (RQ1)

This experiment aims to demonstrate whether the functional redundancy semantic metric can be considered as quantitative measure of program redundancy expressed through the excess output data generated by this program function.

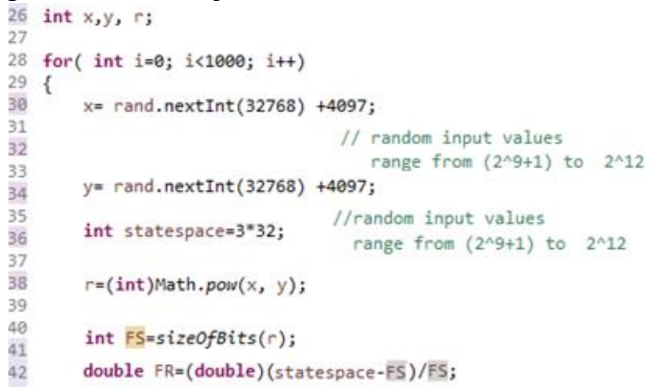
In this experiment, we consider five programs which are the addition of two integers (Sum), the Greatest Common Divisor (GCD), the average of two integers (AVG), the Power program (Power) and the maximum value program (MaxValue). So, to answer the first RQ (RQ1), we proceed as follows:

- We perform 10 different executions for each program. In each execution, we consider 1000 random input values but we change each time the range of these inputs. So:
    - o We start by using 1000 random inputs that range from ($2^{\wedge 1}$+1) to $2^{\wedge 3}$.
    - o We increase the range of inputs by adding in each new execution, 3 new bits. So, in the second execution, the input values will range from ($2^{\wedge 3}$+1) to $2^{\wedge 6}$.
    - o In the third execution, we add 3 other bits, so, the input values will range from ($2^{\wedge 6}$+1) to $2^{\wedge 9}$.
    - o In the fourth execution, 3 other bits are added, the input values will range from ($2^{\wedge 9}$+1) to $2^{\wedge 12}$.
    - o In the fifth execution, 3 other bits are added, the input values will range from ($2^{\wedge 12}$+1) to $2^{\wedge 15}$.
    - o We proceed in the same way for the other executions by adding in each time 3 new bits.

So, for the final execution, the 1000 random input values will range from $(2^{27}+1)$ to 2^{30}.

- For each experiment, we generate the FR metric as well as the used entropy of the program output which is the number of bits used to store the output value. We aim to identify the relationship between these two variables.

The process of the automatic generation of FR metric consists of different steps. The main steps are presented in Figure. 4 that details the different steps for the automatic generation of this metric for the Power program. Concerning the other programs, the same steps are used:



```
26 int x,y, r;
27
28 for( int i=0; i<1000; i++)
29 {
30     x= rand.nextInt(32768) +4097;
31
32                                      // random input values
33                                         range from (2^9+1) to  2^12
34     y= rand.nextInt(32768) +4097;
35                                      //random input values
36     int statespace=3*32;               range from (2^9+1) to  2^12
37
38     r=(int)Math.pow(x, y);
39
40     int FS=sizeOfBits(r);
41
42     double FR=(double)(statespace-FS)/FS;
```

Figure 4. A part of functional redundancy computing for the Power program

This process consists of the following steps:

- First, the state space H(S) of each program as the entropy (bits) of the declared variables is computed. For instance, for the Power program, the state space is computed as shown in lines 36 in Figure.4.
- Second, we use Equation (1) to compute the final state space H(Y) we denote by FS that is the number of bits (entropy) required to store the result of the program' execution. For instance, for the Power program, this value is computed as shown in line 40 in Figure.4.
- Third, the functional redundancy metric is deduced using Equation (1). For instance, for the Power program, it is computed as it is shown in line 42 of Figure. 4. The same steps are used to compute this metric for the other programs considering the variables used for each one.

The structure of the generated data sets for each program is presented in Table 1:

Table 1. Structure of the generated Data sets for experiment 1

Entropy	**FR metric**

The data sets details are available for readers at this link: https://sites.google.com/view/fr-semantic-metric-data-sets/accueil. The results analysis will be performed in section VI.A.

2) Experiment 2: Functional redundancy metric for Error detection (RQ2)

This experiment aims to demonstrate whether the functional redundancy can be considered as an error detection indicator.

In this experiment, we consider two P and P1 programs. P is a correct program on which faults are injected to obtain a new version of this program P1 (see coloured statements).

The injected faults are not related to the program syntax but they have an impact on its functionality:

<u>P (correct version):</u>
{int x, x1, x2, x3, x4, x5;
1 x= rand.nextInt(100) +1;
2 x=x+x1;
3 x=2*x+2*x3;
4 x=x%x1;
5 x=x+x4;
6 x=x*x5;
}

<u>P1 (Faulty version):</u>
{int x, x1, x2, x3, x4, x5;
1 x= rand.nextInt(100)+1;
2 x=x*x1;
3 x=2*x+2*x3;
4 x=x/x1;
5 x=x4+x4;
6 x=x*x5;
}

So, to answer RQ2, we proceed as follows:

- We automatically generate the FR semantic metric for the two P and P1 programs using 1000 random inputs.
- We compare the results of the two programs, the objective is to identify if the FR metric will be different between P and P1 programs.

The structure of the generated data sets is presented in Table 2:

Table 2. Structure of the generated Data sets for experiment 2

FR metric for P (Correct version)	**FR metric for P1 (Faulty version)**

The data sets details are available for readers at this link: https://sites.google.com/view/fr-semantic-metric-data-sets/accueil. The results analysis will be performed in section VI.B.

VI. Data Analysis and Discussion

This section describes the data analysis and discussion for each of the presented experiments.

A. Experiment 1: Functional redundancy metric for redundancy assessment (RQ1)

Based on the generated data sets of the first experiment represented in Table 1, we aim to identify whether the FR semantic metric is useful to assess the excess data (redundancy) of a program output. So, we resort to a statistical methodology and we proceed as follows:

- We start by data standardization: this step is required since the FR and the used entropy variables have different units. So, we use the STATA tool to perform the standardization of our data sets.
- We perform the normality tests to identify the appropriate correlation test to be used. For more details, readers are referred to [15].
- We perform the Spearman correlation test based on the following hypothesis:
    - H0: $\rho = 0$ (null hypothesis) there is no

significant correlation between FR metric and the used entropy.

- H1: ρ ‡ 0 (alternative hypothesis) there is a significant correlation between FR metric and the used entropy.

The results of Spearman correlation for the different programs are presented in Table 3:

Table 3. Structure of the generated Data sets for experiment 1

Entropy	FR metric

Table 3 shows that most of the correlation coefficients are strong and negative for the different programs. This indicates that there is a strong negative correlation between the functional redundancy and the used entropy of the program output. In other words, each time the used entropy increases, the FR metric that measures the redundancy of this program output will tend to decrease. Consequently, it is possible to answer the RQ1 which suggests that the FR measures the redundancy of the program function. These results are also summarized in Figure. 5.

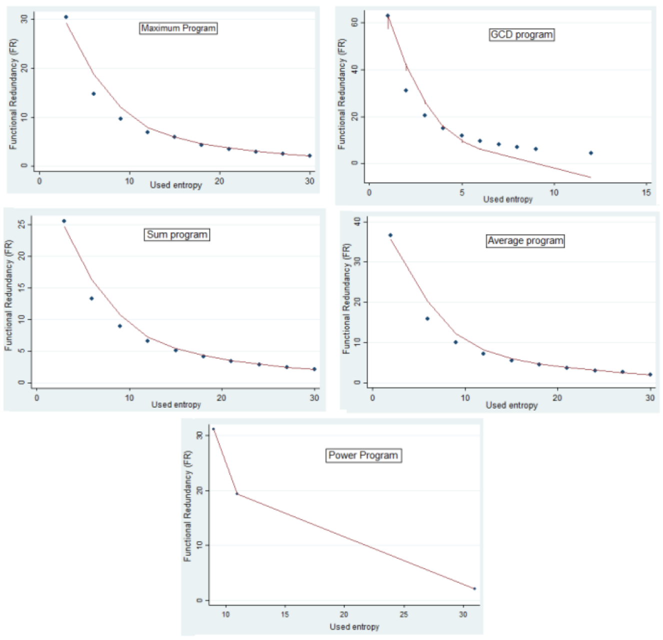


Figure 5. Correlation between FR and the used entropy

Figure. 5 shows that for the different programs, the output redundancy computed by the FR metric as the excess output data generated by these programs function is negatively correlated by the used entropy of this output. So, we state that the FR metric is useful to quantitatively measure programs redundancy.

B. *Experiment 2: Functional redundancy metric for error detection (RQ2)*

To perform the analysis of the generated data sets of the second experiment presented in Table 2, we compare the FR metric for the two P and P1 programs presented in section V.B.2. Results are illustrated in Figure.6:

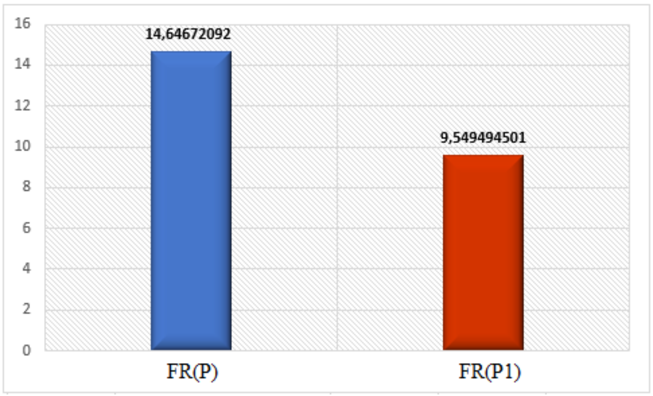


Figure 6. Comparison of FR metric between P and P1 programs

Figure.6 shows that there is a difference between the values of the FR metric between the two programs. So, for the correct program, the value of this metric is 14.64, whereas, for P1, its value is 9.54. This difference is caused by the injected faults. Hence, for developers and testers, the FR metric is useful to indicate whether their programs are error-free. Consequently, it is possible to answer the second RQ (RQ2) and we can note that the FR metric is useful as an error detection indicator.

C. *Overall analysis of the results*

The functional redundancy semantic metric is proposed as quantitative measure of programs redundancy on one side and as an error detection indicator on the other side. The experimental study we performed, confirms these hypotheses. Hence, we stated that the functional redundancy metric is useful to compute the excess output data generated by a program function. This because the FR metric is negatively correlated with the used entropy of this output. Moreover, this metric is useful for developers and testers since it indicates whether their programs are error-free.

VII. Conclusion and Perspectives

The presented study discusses three main concepts which are software fault tolerance, redundancy and software metrics for redundancy assessment.

The literature performed in this paper shows that fault tolerance is one of the quantifiable attributes that helps to achieve reliable and dependable software systems. Most of fault tolerance techniques are based on redundancy. The assessment of redundancy is necessary required to reflect the ability of software programs to tolerate faults. Moreover, considerable attention has been paid to the use of semantic metrics as quantitative measures of programs redundancy.

Among these metrics, the functional redundancy semantic metric is defined to measure programs redundancy as the excess output data generated by a program function. On another hand, this metric is proposed to be used as an error detection indicator based on the assessed programs redundancy. However, the key limitation of this metric is that it is manually computed for procedural programs and only a theoretical basis was presented for it. To solve this problem, we started by proposing an automated way to compute it for different java programs.

Based on a robust statistical approach, we perform the analysis of the generated data sets, different interpretations are identified. First, for developers, an automatic calculation of this metric is necessary required. Additionally, it is an important step that helps to construct an empirical data base of this metrics' values. Then, we demonstrate that the functional redundancy metric is useful as a quantitative measure of programs redundancy. Finally, we demonstrate that this metric is useful as an error detection indicator based on programs redundancy assessment.

Even though the presented benefits presented in this work could be enhanced. As future work, we envision to focus on extending our experiment to incorporate open source java programs and to study the use of these metrics as measures of different quality attributes like defect density.

References

[1] ISO/IEC 25000:2014 Systems and software Engineering — Systems and software product Quality Requirements and Evaluation (SQuaRE)-- Guide to SQuaRE, 4.33

[2] Fenton, N., & Bieman, J. (2014). Software metrics: a rigorous and practical approach. CRC press.

[3] Lyu, M. R. (1996). Handbook of software reliability engineering.

[4] Etzkorn, L., & Delugach, H. (2000). Towards a semantic metrics suite for object-oriented design. In Technology of Object-Oriented Languages and Systems, 2000. TOOLS 34. Proceedings. 34th International Conference on (pp. 71-80). IEEE doi : 10.1109/TOOLS.2000.868960.

[5] Carzaniga, A., Mattavelli, A., & Pezzè, M. (2015, May). Measuring software redundancy. In Proceedings of the 37th International Conference on Software Engineering-Volume 1 (pp. 156-166). IEEE Press.

[6] Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. IEEE Transactions on software engineering, 20(6), 476-493. doi: 10.1109/32.295895.

[7] Mattavelli, A. (2016). Software redundancy (Doctoral dissertation, Università della Svizzera italiana).

[8] Li, W., & Henry, S. (1993, May). Maintenance metrics for the object oriented paradigm. In Software Metrics Symposium, 1993. Proceedings., First International (pp. 52-60). IEEE.

[9] Abreu, F. B., & Melo, W. (1996). Evaluating the impact of OO Design on Software Quality. In Proc. Third Int'l Software Metrics Symp.

[10] Marcus, A., Poshyvanyk, D., & Ferenc, R. (2008). Using the conceptual cohesion of classes for fault prediction in object-oriented systems. IEEE Transactions on Software Engineering, 34(2), 287-300.

[11] Stein, C., Etzkorn, L., Gholston, S., Farrington, P., Utley, D., Cox, G., & Fortune, J. (2009). Semantic metrics: Metrics based on semantic aspects of software. Applied Artificial Intelligence, 23(1), 44-77. doi: 10.1080/08839510802573574.

[12] Arvanitou, E. M., Ampatzoglou, A., Chatzigeorgiou, A., & Avgeriou, P. (2016). Software metrics fluctuation: a property for assisting the metric selection process. Information and Software Technology, 72, 110-124.

[13] Mili, A., Jaoua, A., Frias, M., & Helali, R. G. M. (2014). Semantic metrics for software products. Innovations in Systems and Software Engineering, 10(3), 203-217.

[14] Mili, A., & Tchier, F. (2015). Software testing: Concepts and operations. John Wiley & Sons.

[15] Amara, D., Fatnassi, E., & Rabai, L. (2017, December). An Automated Support Tool to Compute State Redundancy Semantic Metric. In International Conference on Intelligent Systems Design and Applications (pp. 262-272). Springer, Cham.

[16] Amara, D., Fatnassi, E., & Rabai, L. (2018, December). An Empirical Assessment of Functional Redundancy Semantic Metric. In International Conference on Intelligent Systems Design and Applications. Springer, Cham.

[17] Bourque, P., & Fairley, R. E. (2014). Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0. IEEE Computer Society Press.

[18] Boulanger, J. L. (2017). Certifiable Software Applications 4.

[19] Avizienis, A., Laprie, J. C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. IEEE transactions on dependable and secure computing, 1(1), 11-33.

[20] Pullum, L. L. (2001). Software fault tolerance techniques and implementation. Artech House.

[21] Dubrova, E. (2013). Fault-tolerant design (pp. 55-65). New York: Springer.

[22] Rizwan, M., Nadeem, A., & Khan, M. B. (2015, December). An evaluation of software fault tolerance techniques for optimality. In Emerging Technologies (ICET), 2015 International Conference on (pp. 1-6). IEEE.

[23] Lyu, M. R., Huang, Z., Sze, S. K., & Cai, X. (2003, November). An empirical study on testing and fault tolerance for software reliability engineering. In Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on (pp. 119-130). IEEE.

[24] Jaoua, A., & Mili, A. (1990). The use of executable assertions for error detection and damage assessment. Journal of Systems and Software, 12(1), 15-37.

[25] Laprie, J. C. (1992). Dependability: Basic concepts and terminology. In Dependability: Basic Concepts and Terminology (pp. 3-245). Springer, Vienna.

[26] Mili, A., Wu, L., Sheldon, F. T., Shereshevsky, M., & Desharnais, J. (2006, March). Modeling Redundancy: Quantitative and Qualitative Models. In AICCSA (pp. 1-8).

[27] Asghari, S. A., Marvasti, M. B., & Rahmani, A. M. (2018). Enhancing transient fault tolerance in embedded systems through an OS task level redundancy approach. Future Generation Computer Systems, 87, 58-65.

[28] Verma, A., Ghartaan, A., & Gayen, T. (2016). Review of Software Fault-Tolerance Methods for Reliability Enhancement of Real-Time Software Systems. International Journal of Electrical and Computer Engineering (IJECE), 6(3), 1031-1037.

[29] Eckhardt, D. E., Caglayan, A. K., Knight, J. C., Lee, L. D., McAllister, D. F., Vouk, M. A., & Kelly, J. P. J. (1991). An experimental evaluation of software redundancy as a strategy for improving reliability. IEEE Transactions on

software engineering, 17(7), 692-702. doi:10.1109/32.83905.

[30] Jiang, L., & Su, Z. (2009, July). Automatic mining of functionally equivalent code fragments via random testing. In Proceedings of the eighteenth international symposium on Software testing and analysis (pp. 81-92). ACM. doi : 10.1145/1572272.1572283.

[31] Higo, Y., & Kusumoto, S. (2014, November). How should we measure functional sameness from program source code? an exploratory study on java methods. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (pp. 294-305). ACM. doi: 10.1145/2635868.2635886.

[32] Bansiya, J., Davis, C., & Etzkorn, L. (1999). An entropy-based complexity measure for object-oriented designs. Theory and Practice of Object Systems, 5(2), 111-118. doi: 10.1002/(SICI)1096-9942.

[33] Hamming, R. W. (1950). Error detecting and error correcting codes. Bell System technical journal, 29(2), 147-160.

[34] Shannon, C. E. (2001). A mathematical theory of communication. ACM SIGMOBILE mobile computing and communications review, 5(1), 3-55.

[35] Cox, G.W, Gholston S.E., Utley D. R., Etzkorn L.H., Gall C.S, Farrington P.A. and Fortune J.L. Empirical Validation of the RCDC and RCDE Semantic Complexity Metrics for Object-oriented Software, Journal of Computing and Information Technology - CIT 15(2), pp. 151- 160 (2007).

[36] Nuñez-Varela, A. S., Perez-Gonzalez, H. G., Martínez-Perez, F. E., & Soubervielle-Montalvo, C. (2017). Source code metrics: A systematic mapping study. Journal of Systems and Software, 128, 164-197.

[37] Arvanitou, E. M., Ampatzoglou, A., Chatzigeorgiou, A., Galster, M., & Avgeriou, P. (2017). A mapping study on design-time quality attributes and metrics. Journal of Systems and Software, 127, 52-77.

[38] Tahir, T., Rasool, G., & Gencel, C. (2016). A systematic literature review on software measurement programs. Information and Software Technology, 73, 101-121.

[39] Kabir, S. (2017). An overview of fault tree analysis and its application in model-based dependability analysis. Expert Systems with Applications, 77, 114-135.

[40] de AG Saraiva, J., De França, M. S., Soares, S. C., Fernando Filho, J. C. L., & de Souza, R. M. (2015). Classifying metrics for assessing object-oriented software maintainability: A family of metrics' catalogs. Journal of Systems and Software, 103, 85-101.

Author Biographies

Dalila Amara PhD student and member of research team at SMART research laboratory, High Institute of Management, Tunis University, Tunisia. Research interests include software engineering and software quality measurement.

Ezzeddine Fatnassi Assistant professor of computer science, and member of research team at SMART research laboratory. PhD in computer science is received from High Institute of Management, Tunis University, Tunisia in 2015. Research interests include optimization heuristics and software engineering.

Latifa Rabai Professor of computer science, College of Business, University of Buraimi, Al Buraimi, Sultanate of Oman. Member of research team at SMART research laboratory, High Institute of Management, Tunis University, Tunisia. Research interests include software engineering, E-learning, computer security and reliability and cloud computing.