Semantic Metrics for Software Products [*]

A. Mili
College of Computer Science
NJIT
Newark NJ 07102-1982
mili@cis.njit.edu

A. Jaoua
Department of Computer Science
Qatar University
Doha, Qatar
jaoua@qu.edu.qa

M. Frias
Department of Software Engineering
Instituto Tecnologico de Buenos Aires
Buenos Aires, Argentina
mfrias@itba.edu.ar

Rasha Gaffer Mohamed Helali
College of Engineering
SUST
Khartoum, Sudan
Rasha_800@hotmail.com

April 10, 2014

Une Science a l'Age de ses Instruments de Mesure.
A Science is as Advanced as its Instruments of Measurement.
Gaston Bachelard, 1884-1962.

Abstract

Like all engineering disciplines, software engineering relies on quantitative analysis to support rationalized decision-making. Software engineering researchers and practitioners have traditionally relied on software metrics to quantify attributes of software products and processes. Whereas traditional software metrics are typically based on a syntactic analysis of software products, we introduce and discuss metrics that are based on a semantic analysis: our metrics do not reflect the form or structure of software products, but rather the properties of their function.

At a time when software systems grow increasingly large and complex, the focus on diagnosing, identifying and removing every fault in the software product ought to relinquish the stage to a more measured, more balanced, and more realistic approach, which emphasizes failure avoidance, in addition to fault avoidance and fault removal. Semantic metrics are a good fit for this purpose, reflecting as they do a system's ability to avoid failure rather than its proneness to being free of faults.

Keywords

Syntactic metrics, semantic metrics, state redundancy, functional redundancy, error maskability, requirements flexibility.

1 Introduction

1.1 Semantic Metrics

Like all engineering disciplines, Software Engineering relies on quantitative analysis to support decision-making that pertains to the management of products and processes. To this effect, researchers have long been interested in defining and analyzing metrics that capture properties of software products and software processes, to such an extent that software metrics have long since outgrown the laboratory stage and are now the subject of regular textbooks [7, 9, 2], and common industrial practice. Most software metrics in use nowadays (and certainly the most widely known) are based on syntactic

[*]Acknowledgement: This publication was made possible by a grant from the Qatar National Research Fund NPRP04-1109-1-174. Its contents are solely the responsability of the authors and do not necessarily represent the official views of the QNRF.

attributes of software artifacts; as such, they reflect how a program is represented, but not what a program does; yet, many important program attributes may have more to do with the latter than the former. In addition, many software attributes of interest are not intrinsic to the software product, but also involve the specification that the software product is supposed to satisfy; hence if we want metrics to reflect relevant quality attributes, we need to pay attention not only to the software product, but also to its specification. We find it curious that in all the research on the correlation between software metrics on one hand and fault density, fault proneness, and fault forecasting on the other hand, no consideration is ever given to specifications; yet a fault is a fault only with respect to a specification, hence to be comprehensive, software metrics ought to take into account attributes of specifications along with attributes of programs.

In this paper, we introduce a number of software metrics that reflect semantic properties of software products, and are independent of the minute details of how products are represented. In keeping with the E-4 discipline of [7], which is a refinement of Basili's GQM paradigm [14], we proceed in a stepwise manner, as follows:

- *Establish*: In the *Establish* phase of the E-4 paradigm, one needs to define the goals of the metrics. In our case, we are interested to monitor/ control product reliability.
- *Extract*: In the *Extract* phase, one must identify what quantifiable attributes can help to achieve the goals set forth in the previous phase. In our case, we are interested in computing quantitative functions that reflect a program's potential for fault tolerance; in our approach, this involves analyzing the program as well as its specification. Our focus on fault tolerance as a criterion for software quality stems from two premises:
  - In [19], Northrop et. al present the result of a twelve month study conducted at the Software Engineering Institute (CMU, Pittsburgh), with the goal of charting the research agenda of the software engineering discipline, in light of the expected emergence of large scale software systems; referred to as *Ultra Large Scale Systems* (ULS for short), these systems are expected to have sizes in excess of 1 billion lines of code. The ULS panel finds that to control the quality of systems of this scale, we must reason at a macro-level in terms of broad system properties, rather than minute statement-level detail; also, scale and complexity dictate a shift from a focus on fault removal to a focus on failure avoidance.
  - In [20], Patterson and Fox present a joint UC Berkeley/ Stanford project titled *Recovery Oriented Computing*, which argues in favor of controlling software quality through making error recovery a pervasive computing paradigm; rather than straining to find and remove faults in software products (a tedious and increasingly unrealistic aim, in light of increasing size and complexity of software systems), this paradigm advocates providing the system with pervasive means to recover from errors when they arise.

  Whereas correctness can only be established through a painstaking detailed analysis process, fault tolerance can conceivably be achieved by providing the system with high level error detection, damage assessment, and error recovery mechanisms.
- *Evaluate*: In the *Evaluate* phase of the E-4 paradigm, one needs to evaluate the selected metrics to assess their fitness for the goals established in the first phase. We envision two venues to evaluate the fitness of our metrics: an analytical approach, which aims to compute or approximate quality attributes from semantic metrics; and empirical approach, which collects statistical data regarding the link between our semantic metrics and observations of quality in software systems.
- *Execute*: In the *Execute* phase of the E-4 paradigm, one needs to deploy the selected metrics, once they are validated, to help achieve the goals set forth in the first phase. This is part of our future research plans.

We readily acknowledge that because they reflect the semantics of a program rather than its syntax, our metrics fail to capture an important attribute, namely the complexity of the program; indeed, complexity is not a feature of the function of a program as much as it is a feature of its representation. Complexity may be considered an important attribute to quantify, for our purposes, because it is usually correlated with high fault density. But it is only fitting that, because our focus is on failure avoidance rather than fault removal, our metrics should reflect fault tolerance rather than fault proneness.

1.2 Related Work

Semantic software metrics are not a new idea. In [21], Voas and Miller argue that software components that have a high domain size to range size ratio are prone to hide faults, hence ought to be tested more throughly; accordingly, they introduce the concept of *DDR* (*domain to range ratio*) as a software metric, and argue that this is a semantic metric, in the sense that it reflects the semantic properties of a software component rather than how the component is represented in source code. In [16] Morell and Murrill broaden the analysis of Voas and Miller [21] by defining semantic metrics that do not merely capture the input/output behavior of software components, but rather reflect their stepwise execution, and in particular how faults are propagated or masked throughout an execution. In [17], Morell and Voas further elaborate on the ideas of Voas and Miller [21] by presenting a framework that allows them to quantify semantic information of programs, specifically information concerning how program states evolve throughout an execution. All the metrics used by Voas et. al are variations on the non-injectivity metric that we introduce in this paper, applied in the special case where probability distributions are uniform.

Another school of thought on semantic software metrics is based primarily at the University of Alabama at Huntsville, and generates software metrics by analyzing concepts and relations in source code, most notably object oriented code, and inferring quantitative attributes therefrom. In [10], Gall et al discuss the introduction of semantic software metrics that quantify attributes of software artifacts by means of an analysis of their design documentation and their requirements specifications; to this effect, they perform a natural language analysis of these documents in a bid to infer the complexity of a product from its concept and relationship structure. In [3, 8] Etzkorn and her team define entropy-based software metrics for object oriented software products. The proposed metrics are reminiscent of Halstead's software science metrics [11], but rather than measure syntactic tokens, they measure semantic tokens such as concepts and relationsships.

The software metrics that we present in this paper are semantic in the following sense: they view software products as aggregates of spaces, functions and relations; and they reflect the set theoretic properties of theses spaces, functions and relations. We conduct our discussion in the context of C-like procedural programs, but to the extent that other types of programs (object oriented programs, functional programs, logic programs, etc) are built from the same basic components, our discussions may apply to them as well.

1.3 Agenda

In the next section, we briefly present the main mathematical background that we need for this paper, which includes elements of relational mathematics, elementary concepts of information theory, and elementary concepts of software fault tolerance. Then we consider the main phases of software fault tolerance, namely, error detection, error masking (making recovery unnecessary) and error recovery (if recovery proves necessary), and we see what attributes of a software product promote/ facilitate the execution of these phases. In sections 3, 4, 5, we consider these attributes in turn and see how we can quantify them, thereby producing our suite of semantic metrics.

2 Background

2.1 Relational mathematics

Our main source for this section is [5], to which the interested reader is referred, for further details. We consider a set S defined by the values of some program variables, say x, y and z; we typically denote elements of S by s, and we note that s has the form $s = \langle x, y, z\rangle$. We use the notation $x(s)$, $y(s)$, $z(s)$ to denote the x-component, y-component and z-component of s, respectively. We may sometimes use x to refer to $x(s)$ and x' to refer to $x(s')$, when this raises no ambiguity. We refer to elements s of S as *program states* and to S as the *state space* (or *space*, for short) of the program that manipulates variables x, y and z. Given a program g on state space S, we use functions on S to capture the mapping that the program defines from its initial states to its final states, and we use relations on S to capture functional specifications that we may want the program to satisfy. To this effect, we briefly introduce elements of relational mathematics. A relation on S is a subset of the Cartesian product $S \times S$. Constant relations on some set S include the *universal* relation, denoted by L (=$S \times S$), the *identity* relation, denoted by I, and the *empty* relation, denoted by $\emptyset$.

Because relations are sets, we apply the usual set theoretic operations between relations: union ($\cup$), intersection ($\cap$), and complement ($\overline{R}$). Operations on relations also include the *converse*, denoted by $\widehat{R}$, and defined by $\widehat{R} = \{(s, s')|(s', s) \in R\}$. The *product* of relations R and R' is the relation denoted by $R \circ R'$ (or RR') and defined by $R \circ R' = \{(s, s')|\exists s'' : (s, s'') \in R \wedge (s'', s') \in R'\}$. The *domain* of relation R is defined as *dom*$(R) = \{s|\exists s' : (s, s') \in R\}$. The *range* of relation R is denoted by *rng*(R) and defined as *dom*$(\widehat{R})$. We admit without proof that for a relation R, $RL = \{(s, s')|s \in \mathit{dom}(R)\}$ and $LR = \{(s, s')|s' \in \mathit{rng}(R)\}$. The *nucleus* of relation R is the relation denoted by $\mu(R)$ and defined as $R\widehat{R}$. The *co-nucleus* of relation R is the relation denoted by $\gamma(R)$ and defined as $\widehat{R}R$.

We say that relation R is *total* if and only if $\mu(R) = L$ and we say that relation R is *surjective* if and only if $\gamma(R) = L$. Given two relations R and R' that have the same domain, we say thet R is *more-injective* than R' if and only if $\mu(R) \subseteq \mu(R')$ and we say that R is *injective* if and only if it is more-injective than I; the name *more-injective* may be misleading, given that we are talking about a reflexive ordering (it should be more-injective-than-or-as-injective-as), but we adopt it for convenience. Given two relations R and R' that have the same range; we say that R is *more-deterministic* than R' if and only if $\gamma(R) \subseteq \gamma(R')$, and we say that relation R is *deterministic* if and only if it is more-deterministic than I.

2.2 Information Theory

Our main source for this section is [6], to which the interested reader is referred, for further details. Given a variable X on a finite set X (by abuse of notation we use the name to represent the random variable and the set from which the random variable may take its values), we let the *entropy* of X be the following function:

$$H(X) = -\sum_{i=1}^{n} \pi_X(x_i) \log(\pi_X(x_i)),$$

where

- $\log$ is the base 2 logarithm,
- $X = \{x_1, x_2, x_3, ...x_n\}$,
- $\pi_X(x_i)$ is the probability of the event: $X = x_i$.

We admit without proof that $H(X) \geq 0$; also, we take as a convention that the expression $p \log(p)$ equals zero when p equals 0, hence we may apply the entropy function to probability distributions that are not necessarily non-zero for all x_i. Intuitively, the entropy of random variable X represents the amount of uncertainty regarding the outcome of the random variable, and takes its maximal value (which is $\log(n)$) when all the outcomes are equally likely ($\pi(x_i) = \frac{1}{n}$ for all i).

Given two random variables X and Y on sets X and Y, and we let π_X and π_Y be probability distributions of X and Y over their respective sets; we let π_{XY} be the probability distribution of the events $(X = x_i \wedge Y = y_j)$ over the Cartesian product $X \times Y$. Then we denote by $H(X, Y)$ the entropy of the aggregate random variable (X, Y) over the set $(X \times Y)$, and we refer to it as the *joint entropy* of X and Y. Using this definition, we let the *conditional entropy* of X with respect to Y be denoted by $H(X|Y)$ and be defined as follows:

$$H(X|Y) = H(X, Y) - H(Y).$$

Whereas the entropy of X represents the amounts of uncertainty about the outcome of X, the conditional entropy of X with respect to Y represents the amount of uncertainty about the the outcome of X once we know the outcome of Y. We have an identity to the effect that the joint entropy of (X, Y) is greater than or equal to the entropy of Y, hence the conditional entropy is non-negative.

Given a random variable X that takes its values in some space S, and given a function G on S, we let Y be the random variable $Y = G(X)$, whose probability distribution is derived from that of X, i.e.

$$\pi_Y(Y = y) = \sum_{\forall x: G(x) = y} \pi_X(X = x).$$

Then, we have the inequality ([6]): $H(X) \geq H(Y)$. In other words, applying a function to a random variable reduces its entropy (due to posible loss of information). If G is total and injective, then $H(G(X)) = H(X)$.

To conclude this section, we introduce a concept that we use throughout this paper to assign intuitive interpretations to our metrics.

Definition 1 *We consider a set S and a predicate A on S, and we let S_A be the subset of S defined by elements of S that satisfy $A(s)$. The bandwidth of assertion A is defined as $H(S) - H(S_A)$.*

Consider a set S defined by three integer variables, say x, y and z. Under the hypothesis of uniform probability distribution, and assuming that integers are represented by 32-bit words, the entropy of S is 96 bits. We consider a number of possible assertions, and compute their corresponding bandwidths:

- We define $A(s)$ as $x = y$. Then space S_A is defined by variables y and z only. The entropy of S_A under the hypothesis of uniform probability distribution is $H(S_A) = 64$ bits, hence the bandwidth of A is 32 bits, which is the width of the two expressions (x and y) involved in assertion A.
- We define $A(s)$ as $x = z \wedge y = z$. Then space S_A can be defined by a single variable, say z. The entropy of S_A under the hypothesis of uniform probability distribution is $H(S_A) = 32$ bits, hence the bandwidth of A is 64 bits, which is the combined width of the expressions that are involved in assertion A.
- We define $A(s)$ as $x = 0 \wedge y = 10 \wedge z = 20$. Then space S_A is a singleton, whose entropy is zero, hence the bandwidth of A is 96 bits, which is the combined width of the expressions that are involved in assertion A.

As another brief example, consider the binary representation of ascii characters in a byte; seven bits out of eight are used to represent data, and the eighth bit is used for parity checking. We let S be the set of 8-bit patterns and we let A be the parity test, which can be written as

$$parity(b1..b7) = b8.$$

The bandwidth of this assertion is $H(S) - H(S_A)$, which is $8 - 7 = 1$ bit. Indeed, assertion A is an equality between two 1-bit expressions.

2.3 Fault Tolerance Methodology

Our main source for this section is [13], to which the interested reader is referred, for further details. We consider a program g on some space S, of the form

```
    g = {g1;  L: g2;}
```

where $g1$ and $g2$ are subprograms and L is a label preceding $g2$. We let R be a relation on S that represents the specification that g must meet, and we let s_0 be an arbitrary initial state of g.

- A *fault* in program g is a feature of g that precludes it from satisfying its specification (in the sense of [15], for example).
- An *error* of the program at label L for initial state s_0 is a state that is distinct from the expected state at this label; a fault may or may not cause a fault at label L, depending on the initial state s_0; when a fault does cause an error, we say that it has been *sensitized* by the initial state s_0.
- A *failure* of program g occurs whenever the error that arises at label L causes the program to fail to produce a correct (with respect to R) final state for initial state s_0. An error at label L may cause a failure of the program, in which case we say that the error has been *propagated*; it may also cause no failure, in which case we say that the error has been *masked*.

We say that program g is *fault tolerant* if and only if it has provisions for avoiding failure after faults have caused errors. We consider three phases in the fault tolerance process:

- *Error Detection*, when the program detects an inconsistency that indicates that the program state is erroneous.
- *Damage Assessment*, when the program analyzes the current state to determine whether it is maskable (in which case recovery is unnecessary) or recoverable (in which case recovery is necessary and sufficient) or unrecoverable (in which case recovery is insufficient).
- *Error Recovery*, when a recovery is invoked to map the recoverable state into a maskable state and let the computation resume from label L.

As an illustration, consider the space S defined by a natural variable, let the specification be relation R defined by

$$R = \{(s, s') | s' \textbf{ mod } 3 = s^2 \textbf{ mod } 3\},$$

and let g be the program

```
g = {read(s); s=2*s; L: s = s mod 6; write(s);}
```

The intent of the programmer was for g to compute the following function:

$$G = \{(s, s') | s' = s^2 \mathbf{\ mod\ } 6\},$$

which would have been correct with respect to R (in the sense of [15]), since G and R are both total, and $G \subseteq R$, as shown below:

$$s' = s^2 \mathbf{\ mod\ } 6 \Rightarrow s' \mathbf{\ mod\ } 3 = (s^2 \mathbf{\ mod\ } 6) \mathbf{\ mod\ } 3 = s^2 \mathbf{\ mod\ } 3.$$

But the programmer wrote the statement `s = 2*s` instead of the statement `s=s*s`, creating a fault. This fault may or may not be sensitised, depending on the input value.

- For $s_0 = 2$, the fault is not sensitized, since the expressions `2*s` and `s*s` return the same value for $s = 2$.
- For $s_0 = 6$, the fault is sensitized, causing an error ($s = 12$ rather than $s = 36$ at label L), but the error is subsequently masked (since $12 \mathbf{\ mod\ } 6 = 36 \mathbf{\ mod\ } 6$ at the end of the program).
- For $s_0 = 3$, the fault is sensitized, leading to an error ($s = 6$ instead of $s = 9$ at label L); the error is subsequently propagated, causing a failure ($s = 0$ instead of $s = 3$ in the final state); in this instance, program g failed to behave according to its intended function G, but did not fail with respect to its specification R, since $s_0^2 \mathbf{\ mod\ } 3 = 9 \mathbf{\ mod\ } 3 = 0 = 0^2 \mathbf{\ mod\ } 3$; hence, strictly speaking, it satisfies its specification for $s_0 = 3$.
- Finally, for $s_0 = 4$, the fault is sensitized, leading to an error (the state at label L is $s = 8$ rather than $s = 16$); this error is propagated, leading to a final state that is distinct from the expected final state (the output is $s = 2$ rather than $s = 4$); this final state violates the specification, since $2 \mathbf{\ mod\ } 3 \neq 4 \mathbf{\ mod\ } 3$; in this case, the program failed to compute the expected final state, and also failed to satisfy the specification of the program.

The same fault may cause different chains of events, depending on the input.

In order to be fault tolerant, a program must make provisions for error detection (to recognize when the potential of a failure may arise), error masking (to limit cases when recovery is necessary), and error recovery (to map a recoverable state into a maskable state, and let the computation proceed). In the next three sections we introduce three metrics that reflect to what extent a program is likely to support these three capabilities; in section 5, we introduce a metric that reflects to what an extent a specification tolerates that programs deviate from their intended output.

3 Error Detection: Redundancy

Broadly speaking, redundancy is the property of using more data than is needed to represent some information. Whereas redundancy is usually defined in terms of duplicating elements of data (bits, words, etc), we model it instead as an algebraic property of the representation function, i.e., the function that maps information onto data. We distinguish between two types of redundancy in a program: state redundancy and functional redundancy.

3.1 State Redundancy

Given a program g on space S, it is fair to say that in general, not all elements of S represent valid program states. For example, if we need to define a variable to represent the age of an employee, we typically use the type *integer*, even though we actually use a limited range of integers, say between 0 and 120; also, if we want to record the birth date of an employee and her/his age, along with today's date, then we have the identity that the sum of the birth year and the age in years equals the current year. We introduce a *representation relation*, say ρ, which maps valid program states into their representation in S. The simplest representation relations are those that are

- total (each state value has at least one representation),
- deterministic (each state value has at most one representation),
- injective (different states have different representations), and
- surjective (all representations represent valid states).

Not all representation functions satisfy these four properties —in practice hardly any satisfy all four, in fact.

- When a representation relation is not total, we observe a *partial representation* (for example not all integers can be represented in computer arithmetic).
- When a representation relation is not deterministic, we observe an *ambivalent representation*. Consider the representation of signed integers between -7 and +7 using a sign-magnitude format; zero has two representations, -0 and +0 [12].
- When a representation relation is not injective, we observe *loss of precision* (for example, real numbers in the neighborhood of a representable floating point value are all mapped to that value).
- When a representation relation is not surjective, we observe *redundancy* (for example, in a parity-bit representation of characters, not all bit patterns represent legitimate characters).

For the purposes of our discussions, we equate redundancy with non-surjectivity; for the sake of simplicity, we limit our discussion to representation relations that are deterministic, total, and injective —whence each state value has exactly one representation (by virtue of totality and determinacy) and different state values have different representations (by virtue of injectivity). Under this assumption, we refer to representation relations as representation *functions*.

We are interested to quantify the redundancy of the state of a program. To this effect, we need to distinguish between the *actual* state space of the program, which we define as the set of states that the program may be in, and the *declared* state space of the program, which is the set of values that the declared program variables may take. We let ρ be the function that maps each actual state onto its representation as an aggregate of values of the declared variables. We define the state redundancy of the program state by means of the representation function, as follows.

Definition 2 *Let g be a program, and let Σ be the set of actual states of g, and S be the set of declared states of g. If we let ρ be the representation function that to each actual state σ assigns its representation in S, then we define the redundancy of ρ as:*

$$\kappa(\sigma) = H(S) - H(\rho(\sigma)).$$

By virtue of section 2.2, we know that if ρ is total, deterministic and injective, then $H(\rho(\sigma))$ is equal to $H(\sigma)$; hence, when the representation function is total and injective, its redundancy can be written as:

$$\kappa(\rho) = H(S) - H(\sigma).$$

Typically, the set of declared states is fixed for a given program block (which is the scope of typical variable declarations), but the set of actual states varies as the program proceeds through its execution; hence the redundancy of a state representation may vary from one step to the next through the execution of a program. The following Proposition provides that the state redundancy of a program increases as the program proceeds from one state to the next.

Proposition 1 *Let g be a deterministic program on space S and let σ be a state of the program at some step in the execution of g, and σ' be a subsequent state. Then the state redundancy of program g at state σ' is greater than or equal to the state redundancy of state σ.*

Proof. Since g is deterministic, (σ') is obtained from (σ) by application of a function. We have seen in section 2.2 that application of a function to a random variable decreases or preserves the entropy, hence $H(\sigma') \leq H(\sigma)$. Whence, $\kappa(\sigma') \geq \kappa(\sigma)$. **qed**

Hence as the program proceeds from its initial state to its final state, state redundancy increases monotonically with each state transformation; an adequate representtion of the state redundancy of a program is the interval defined by the state redundancy of its initial state, and the state redundancy of its final state. Each of these bounds has an interesting interpretation:

- The state redundancy of the initial state reflects the gap between the minimal bandwidth required to store the program state and the actual bandwidth reserved to that effect. The programmer has some latitude to control this quantity by trying (or not trying) to codify the state of the program in as few variables as possible.
- The state redundancy of the final state reflects the maximum bandwidth of relationships that hold between program variables as a result of the execution of the program.

Whence the following definition.

Definition 3 *Let g be a program, and let Σ be the set of actual states of g, and S be the set of declared states of g. We let the* state redundancy *of program g be denoted by $\kappa(g)$ and defined as the interval*

$$\kappa(g) = [\kappa(\sigma_I)..\kappa(\sigma_F)],$$

where σ_I is the program's initial state and σ_F is its final state.

By abuse of notation, we use the same symbol (κ) to repreesent the state redundancy of a state representation, and the state redundancy of a program. According to the formula of this definition, the redundancy of the states of the program evolves through the interval as the execution of the program proceeds from the initial state to the final state.

As an illustration of this definition, we consider a simple program that reads two integers included between 1 and 1024 and computes their greatest common divisor.

```
{int x, y; cin << x << y;
 //  initial state
 while (x!=y) {if (x>y) {x=x-y;} else {y=y-x;}}
 //  final state
 }
```

The declared state space of the program includes two integer variables, which we assume to be of width 32 bits; hence we find

$$H(S) = 2 \times 32\, bits = 64\, bits.$$

As for σ_I, it consists of two integer values ranging between 1 and 1024; hence we find

$$H(\sigma_I) = 2 \times \log(1024)\, bits = 20\, bits.$$

We derive the state redundancy of the initial state as:

$$\kappa(\sigma_I) = 44\, bits.$$

For the final state, the declared state space is the same, but the actual range of states is now reduced to a single value between 1 and 1024, since variables x and y are identical. Hence we find:

$$\kappa(\sigma_F) = 64\, bits - 10\, bits = 54\, bits.$$

The state redundancy of this program is the following interval:

$$\kappa(g) = [44\, bits..54\, bits].$$

3.2 Functional Redundancy

Whereas state redundancy reflects the excess data in the representation of a state, and can be used to check consistency conditions within the variables of a state, functional redundancy reflects the excess output data generated by a program function, and can be used to check (partially or totally or multiply) whether the function has executed properly. Whereas the redundancy of a state is equated with the non-surjectivity of the representation function (mapping actual states to their representation), the functional redundancy of a program is equated with the non-surjectivity of the program function (mapping initial states to final states, or inputs to outputs).

Definition 4 *We consider a program g on space S, and we let G be the function defined by g on S. We let S be a random variable that takes its values in set S, and we let Y be a random variable that takes its values in the range of G. The* functional redundancy *of program g is denoted by $\phi(g)$ and defined by:*

$$\phi(g) = \frac{H(S) - H(Y)}{H(Y)}.$$

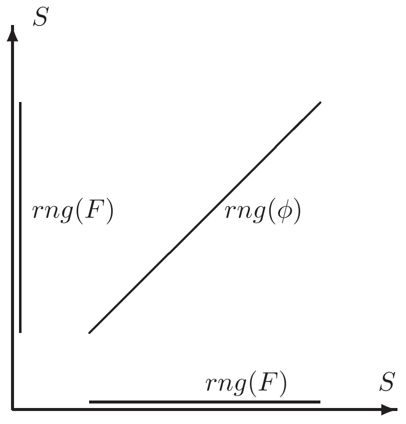

Figure 1: Increasing the Output Space, Preserving the Range

Intuitive Interpretation: The functional redundancy of a program g is the ratio of the excess information that represents the output of g prorated to the entropy of the output produced by g. The funcctional redundancy of a program g may be used to check (partially or totally) the correctness of the output produced by the program, or even to generate the correct output (past the value of 2, through TMR voting for example). So, if $\phi(g) = 0$, there is no scope for checking any property; if $0 < \phi(g) < 1$ then we can check part of the result against redundant information; if $\phi(g) > 0$, then $H(G(S)) \times \phi(g)$ represents the bandwidth of assertions that may be checked on the functional proerties of G. For example, if program g computes the values of five integers, and $\phi(g) = 0.2$, then there may be sufficient redundancy to check that one of the five values is computed correctly. Triple modular redundancy [1] corresponds to a functional redundancy value of 2; more generally, N-ary modular redundancy corresponds to a functional redundancy value of $(N - 1)$. Again, knowing the value of $\phi(g)$ does not tell us how to use the redundant information; but if we can identify it and use it, it can tell us whether we are using all the available redundant information.

Note that when we duplicate, triplicate, or otherwise multiplicate a function (as is done traditionally to enhance error detection and error correction capability), we increase the size (whence, typically, the entropy) of the output space, while preserving the size and entropy of the range of F. Specifically, if we take a function F on S and duplicate it, we obtain a function, say Φ, which has the same input space as F, and the same domain as F. However, the output space of Φ is S^2, and the range of Φ is

$$\{< s, s > | s \in \textit{rng}(F)\},$$

which is homomorphic to *rng*(F). See Figure 1.

To illustrate the significance of this formula, we consider the table of Figure 2. We let the reader contemplate in what sense the relations observed in this table reflect one's intuition about the properties one wants to see in a redundancy function. In this table, we denote by $B5$ (stands for 5-bit integers) the set defined by:

$$B5 = [0..31].$$

Note that all the calculations made in table 2 assume a uniform probability distribution on the domain of F; if we had a non-uniform distribution, we would have a larger value of redundancy. Note also a relation that holds between the redundancy of a function and the redundancy of its duplication:

$$\phi(\langle F, F\rangle) = 2 \times \phi(F) + 1.$$

One way to interpret this is to observe that when one duplicates function F, one duplicates the redundancy of F, plus one copy of the core information of F. Another posible interpretation can be made by rewriting the formula as follows:

$$\phi(\langle F, F\rangle) = 2 \times (\phi(F) + 1) - 1.$$

The latter formula can be interpreted as follows: the redundancy of the duplicate function equals the total information carried out by each instance of the function (which is the redundancy plus 1), minus 1 (to account for the fact that we are counting *excess* information). Note also (by comparing F_3 and F_4, for example) that as the range of the function increases (from 0..3 to 0..15) its redundancy decreases (less surjectivity).

Name	Expression	Input Space	Output Space	Redundancy	Comments
F_1	X	$B5$	$B5$	0	All 5 bits are used
F_2	$2 \times X$	$B5$	$B5$	0.25	Rightmost bit always 0
F_3	$X \ \mathbf{mod}\ 4$	$B5$	$B5$	1.5	Two bits of information, three bits of redundancy
F_4	$X \ \mathbf{mod}\ 16$	$B5$	$B5$	0.25	Four bit of information, one bit of redundancy
F_5	$X\ div\ 2$	$B5$	$B5$	0.25	4 bits of information, leftmost bit is always 0
G_1	$\langle F_1, F_1 \rangle$	$B5$	$B5^2$	1.0	$1 + 2 \times \phi(F_1)$
G_2	$\langle F_2, F_2 \rangle$	$B5$	$B5^2$	1.5	$1 + 2 \times \phi(F_2)$
G_3	$\langle F_3, F_3 \rangle$	$B5$	$B5^2$	4.0	$1 + 2 \times \phi(F_3)$
G_4	$\langle F_4, F_4 \rangle$	$B5$	$B5^2$	1.5	$1 + 2 \times \phi(F_4)$
G_5	$\langle F_5, F_5 \rangle$	$B5$	$B5^2$	1.5	$1 + 2 \times \phi(F_5)$

Figure 2: Functional Redundancy

4 Error Masking: Program Non-Injectivity

Whereas state and functional redundancy enable us to detect errors, maskability enables us to mask them, i.e. produce a subsequent state that bears no trace of the error. What makes this possible in practice is the non-injectivity of programs, i.e. their ability to map distinct states into a single image. The folllowing definition offers a way to quantify the non-injectivity of program functions.

Definition 5 *Let g be a program on space S, whose function is G. Let X be a random variable that takes its values in the domain of G and let Y be defined as $Y = G(X)$. The* non-injectivity *of program g is denoted by $\theta(g)$ and defined by:*

$$\theta(g) = H(X|Y).$$

We have observed in section 2.2 that conditional entropies are non-negative, hence $\theta(g) \geq 0$.

To justify this definition, we proceed in two steps: First, we assume a uniform probability distribution over variable X; then the entropy of X given Y measures the amount of uncertainty we have about the initial state of g if we know its final state; this quantity is a natural representation of non-injectivity, in the sense that the more initial states map to the same image, the bigger the entropy. Second, we consider the question: why does a non-uniform probability distribution represent smaller non-injectivity? The answer is that with a non uniform probability distribution, fewer possible input values have a higher probability of occurrence, culminating in a smaller set of inputs mapping to a single output, hence a less injective behavior.

Intuitive Interpretation: The non-injectivity of program g is expressed in Shannon bits and represents the bandwidth of error that the program can potentially mask. For example, if the program handles integer variables of width w each, and the non-injectivity of g is w bits, the program may potentially mask the loss of an integer varible; for the same amount of injectivity, the program may also recover from the violation of an assertion whose bandwidth is w (e.g. an equality between two integer expressions); if the non-injectivity is $2w$, the program can potentially mask the loss of two integer variables; etc. Knowing the value of the program's non injectivity does not tell us what variables may be lost, nor which assertion may be violated, but gives us some indication of the magnitude of error that can be masked without outside intervention.

Proposition 2 *Let g be a program on space S, whose function is G. Let X be a random variable that takes its values in the domain of G and let Y be defined as $Y = G(X)$. The* non-injectivity *of program g can be written as:*

$$\theta(g) = H(X) - H(Y).$$

Proof. According to [6], $H(X|Y) = H(X,Y) - H(Y)$, where $H(X,Y)$ is the joint entropy of X and Y. Given that we consider deterministic programs, Y is a function of X, hence $H(X,Y) = H(X)$. **qed**

We illustrate the concept of non-injectivity by means of some simple examples. Let us consider a program g on space S defined by three integer variables, say i, j and k.

- If g = {`i=i+1;`}, then $\theta(g) = 0$. Ignoring the possibility of overflow, this program is injective, hence $H(X) = H(Y)$. If the state prior to execution of g is erroneous, so will the state after execution of g.
- If g = {`i=j+k`}, then $\theta(g) = w$, where w is the breadth of an integer variable, since $H(X) = 3w$ and $H(Y) = 2w$. This means that program g has the potential to mask the loss of an integer variable. Indeed it does: if i had the wrong value prior to execution of g, then that error will be masked by g.
- If g = {`i=0; j=1; k=10;`}, then $\theta(g) = 3w$, where w is the width of an integer, since $H(X) = 3w$ and $H(Y) = 0$. This means that program g has the potential to mask the loss of three integer variable. Indeed it does: if i, j and k had the wrong values prior to execution of g, then these errors will all be masked by the time g has executed.

In practice, we need to derive rules that allow us to compute the non-injectivity of a program by analyzing its source code. The rules would proceed by induction on the structure of the program, with one rule for each program construct. As an example, we present below the rule for the sequence.

Proposition 3 *The non-injectivity of a sequence of programs is the sum of their non-injectivities:*

$$\theta(g_1; g_2) = \theta(g_1) + \theta(g_2).$$

Proof. We let X, Y, and Z be the random variables representing the state of the program before g_1, between g_1 and g_2, and after g_2. We have: $\theta(g_1) = H(X) - H(Y)$, and $\theta(g_2) = H(Y) - H(Z)$, hence $\theta(g_1) + \theta(g2) = H(X) - H(Y) + H(Y) - H(Z)$, which simplifies to $(H(X) - H(Z)$, which is $\theta(g_1; g_2)$. **qed**

The base case of the inductive process is the assignment statement, which can be handled as shown in the examples above. The most challenging inductive rule would be the iteration rule, for which we envision to use the concept of *invariant relations* introduced in [18]; this matter is currently under investigation.

Whenever we know the function of a program, we can compute its non-injectivity without going through the inductive statement-by-statement analysis, as we illustrate in the following example. We consider a sorting program that sorts integer arrays of size N. For the sake of simplicity, we assume that the space of this program is limited to the arrays (no index variables, booleans, etc). We also assume that all initial arrays are equally likely to occur, hence we have a uniform probability over X. As for Y, it is the set of sorted arrays. We find:

$\theta(g)$
= $\{$by Proposition 2$\}$
$H(X) - H(Y)$
= $\{$by virtue of the hypothesis of uniformity$\}$
$\log(|X|) - \log(|Y|)$
= $\{$arithmetic$\}$
$\log(\frac{|X|}{|Y|})$
= $\{$There are $N!$ random permutations for each sorted array$\}$
$\log(N!)$.

This expression is known to be approximated by $N\log(N) - N$ for N sufficiently large; for $N = 1024$, for example, we find $\theta(G) = 1024 \times 9 = 9216$ bits of maskable error bandwidth. In effect, the types of errors that a sorting routine can mask are all the errors whereby an array is permuted without loss of original values.

In section 3.1, we have found that state redundancy (which measures error detection capability) increases as the program proceeds from one state to the next. In this section, we find that non-injectivity (which measures masking capability) decreases as the program proceeds through its execution.

Proposition 4 *Let program g on space S be a sequence of subprograms g_1, g_2, ... g_n, and let Γ_i, for $1 \leq i \leq n$ be defined as follows:*

$$\Gamma_i = \{g_i, g_{i+1}, ...g_n\}.$$

Then,

$$\forall i : 1 \leq i < n : \theta(\Gamma_i) \leq \theta(\Gamma_{i+1}.$$

Proof. This Proposition stems readily from Proposition 2, by observing that Γ_i can be written as $\Gamma_i = \{g_i; \Gamma_{i+1}\}$. By virtue of Proposition 2, we have: $\theta(\Gamma_i = \theta(g_i) + \theta(G_{i+1})$. From the discussions of section 2.2, we know that $\theta(g_i)$ is non-negative. **qed**

The interpretation of this Proposition is straightforward: as the program proceeds from one state to the next, it has fewer and fewer future functions to apply, hence fewer and fewer options to mask possible errors. In conjunction with Proposition 1, this Proposition shows an interesting dilemma about successive states generated as the program proceeds through its execution: As the program proceeds from one state to the next, it becomes increasingly easy to detect errors (due to increasing state redundancy), but increasingly difficult to mask them (due to decreasing non-injectivity of remaining program functions).

5 Error Recovery: Specification Flexibility

As we saw in section 2.3, a program may fail to compute its intended function and yet still behave according to the specification it is intended to satisfy. In this section, we wish to quantify the amount of flexibility that a specification allows; we present the following definition.

Definition 6 *We consider a specification R under the form of a binary relation on some space S, and we let X be a random variable that takes its values in the domain of R and Y be a random variable that takes its values in the range of R in such a way as to maintain the condition $(X, Y) \in R$. The* non-determinacy *of specification R is denoted by $\chi(R)$ and defined by:*

$$\chi(R) = H(Y|X).$$

A specification is all the more non-deterministic (flexible) that the conditional entropy of its output states for a given input state is greater; bigger entropies are equated with larger sets of possible outputs, and more uniform probability distribution of the occurrence of these outputs.

Intuitive Interpretation: the non-determinacy of a specification is expressed in Shannon bits and represents the bandwidth of deviation of candidate programs from their intended function that does not violate the specification. For example, if state S includes integer variables of width w and we find that the non-determinacy of R is w, then we can lose up to one integer variable and still satisfy the specification.

As an illustrative example, we consider the following specification on space S defined by three integer variables, say i, j, and k.

$$R = \{(s, s')|k = 2i + j \wedge i' = i + j \wedge j' = i - j\}.$$

We let X be a random variable that ranges over the domain of this relaton (i.e. the set of states such that $k = 2i + j$) and we let Y be a random variable that takes its values in the range of this relation, in such a way as to maintain the relation $(X, Y) \in R$. We must compute the non-determinacy of this specification using the formula:

$$\chi(R) = H(X, Y) - H(X).$$

We observe that the inverse of R is deterministic, since it can be written as:

$$\widehat{R} = \{(s, s')|i' = \frac{i+j}{2} \wedge j' = \frac{i-j}{2} \wedge k' = \frac{3i+j}{2}\}.$$

Hence X is a function of Y, and $H(X, Y) = H(Y)$. So that the non-determinacy of R can be written as:

$$\chi(R) = H(Y) - H(X).$$

Assuming uniform probability distribution, we find $\chi(R) = 3w - 2w = 1w$, which means that candidate programs may lose one integer variable and still satisfy specification R; indeed, specification R does not dictate any final value for variable k, allowing candidate programs to lose that variable without violating the specification.

As a second example, we consider the specification that we had introduced in section 2.3. This specification is defined on space S of natural variables, and is defined by:

$$R = \{(s, s')|s' \mathbf{mod}\, 3 = s^2 \mathbf{mod}\, 3\}.$$

We note that the domain of R and the range of R are both equal to S, and we let X and Y be random variables that range over S in such a way as to maintain the property:

$$Y \mathbf{mod}\, 3 = X^2 \mathbf{mod}\, 3.$$

We compute the non-determinacy of relation R using the expression:

$$\chi(R) = H(X, Y) - H(X),$$

using the uniform probability distribution of X and Y. We find, $H(X, Y) = 2w - \log(3)$, and $H(X) = w$. Hence,

$$\chi(R) = w - \log(3)$$

assuming of course that $w \geq 2$, else the program cannot compute the remainder of the division by 3. Assuming that the width of a natural number is 32 bits, this formula finds that the non-determinacy of this relation is 30.415 bits, which is very high considering that the output is an integer of 32 bits. But consider the vast latitude that this specification offers: if s' is a correct final state, then so are $s' + 3k$ for any k.

In general, we need to find a way to compute the non-determinacy of a specification by induction on the structure of the specification, much in the same way as we envision to compute the non-injectivity of a program (re: section 4). But whereas for non-injectivity we do induction on program structures (sequence, alternation, conditional, iteration) using assignment statements for the basis of induction, for non-determinacy we envision to use induction on specification structures (intersection, union, lattice operators [4]) using elementary relations as a basis of induction.

6 An Illustrative Example

We illustrate the foregoing metrics on a simple example, namely the following sorting program.

```
#include <iostream>
#include "rand.cpp"

using namespace std;

//  constants
int N = 100;
int Maxval = 400;

//  functions
void loaddata ();
void sort();
void flusha();

//  state variables
int a[100];

int main ()
{
   loaddata();
   sort();
   flusha();
}

void loaddata()
   {SetSeed(400);
    for (int k=0; k<N; k++) {a[k]=1+Maxval*NextRand();}
   }

void sort()
   {int c, d, p, swap; c=0;
    while (c<(N-1))
       {p=c; d=c+1;
        while (d<N)
           {if (a[p]>a[d]) {p=d;} d++;}
        if (p!=c) {swap=a[c]; a[c]=a[p]; a[p]=swap;} c++;
       }
   }

void flusha()
```

```
  {for (int k=0; k<N; k++) {cout << a[k] << endl;}
  }
```

Specifically, we are interested to compute the semantic metrics of the sort function.

6.1 State Redundancy

As we recall from Definition 3 the state redundancy of a program is the interval bounded by $H(S) - H(\sigma_I)$ and $H(S) - H(\sigma_F)$. To compute $H(S)$, we assume that we are dealing with 32-bit integers, and we count an array of 100 integers and four integer variables, to a total of $104 \times 32 = 3328$. To compute the entropy of the initial state, we observe that we have 101 variables whose value ranges between 1 and 400 (100 cells of the array + swap), and three index variables, whose value ranges between 0 and 99; hence the entropy of the initial state is: $101 \times \log(400) + 3 \times \log(100) = 893\, bits$. Hence the state redundancy of the initial state is: 3328-893=2435 bits.

To compute the state redundancy of the final state, we must assess its entropy, which in turn requires that we estimate the entropy of a sorted array. To this effect, we briefly discuss the question in general terms then apply it specifically to our case study: we know that the entropy of a random array of N cells which range over 2^W values is $N \times W$; the question is, by how much is this entropy reduced when we sort the array. To answer this question, we use an approximation: we divide the range of values into N intervals of equal length and we assume that each cell of the array ranges over the interval of the corresponding rank. The entropy of such an array is then:

$$N \times \log\left(\frac{2^W}{N}\right),$$

which comes out to

$$N \times W - N \log(N).$$

In other words, sorting an array of size N reduces its entropy by $N \log(N)$. In our case, we find that the entropy of the sorted array is: $100 \times \log(400) - 100 \times \log(100) = 100 \times \log(4) = 200\, bits$. Because the values of variables c and d at the final state are determined by the program (to be equal to N), they do not add to the entropy; we only count the entropy of variables p (range: 100 values) and swap (range: 400 values). Hence the entropy of the final state is: $200 + \log(100) + \log(400) = 213\, bits$. Hence the state redundancy of the final state is: 3328-213 = 3115 bits. Whence the state redundancy of this sorting porgram is the following interval:

$$\kappa(sort) = [2435\, bits..3115\, bits].$$

6.2 Functional Redundancy

According to Definition 4, the functional redundancy of this program is given by the following formula:

$$\phi(sort) = \frac{H(S) - H(Y)}{H(Y)},$$

where Y is a random variable that takes its values in the range of the program's function. From the previous subsection, we know that the entropy of S is $H(S) = 3328\, bits$. Also, random variable Y takes its values in the set of final states of the program, whose entropy we have estimated in the previous subsection as: $H(\sigma_F) = 213\, bits$. Hence the functional redundancy of this program is:

$$\phi(sort) = \frac{3328 - 213}{213} = 14.62.$$

This may look very large for a program that has no evidence of massive functional redundancy; but we have to remember that functional redundancy reflects the non-surjectivity of the program function; most of the redundancy that we are observing in this program comes from using integer variables that can represent 2^{32} different values to merely represent 400 different values. If array cells ranged over 2^{32} values, we would have found a functional redundancy of about 0.3.

6.3 Non Injectivity

According to Definition 5, the non-injectivity of this program is given by the following formula,

$$\theta(g) = H(X|Y),$$

where X is a random variable that ranges over the domain of this program's function, and Y is a random variable that takes its values over its range; this condition entropy measures how much do we know about X if we observe Y. Because this program sorts arrays, we know that each ordered array of size N observed at the final state corresponds to $N!$ possible arrays at the initial state. Under the hypothesis of uniform probability, the conditional entropy is:

$$H(X|Y) = \log(N!).$$

According to Stirling's approximation, this quantity can be written as:

$$N \times \log(N) - N = 564\, bits$$

for $N = 100$.

6.4 Non Determinacy

Whereas so far we have computed our semantic metrics by analyzing the program, the metric of non-determinacy depends on the specification against which the program's behavior is judged (to determine correctness, expose failures, diagnose faults, etc). We consider three possible specifications, and estimate the non-determinacy of each.

- Ord, which provides that the final array is sorted (say, in increasing order).
- Prm, which provides that the final array is a permutation of the initial array.
- $Sort = Prm \cap Ord$.

According to Definition 6, the non-determinacy of a specification is given by the formula

$$\chi(R) = H(Y|X),$$

where X is a random variable that ranges over the domain of the specification ($dom(R)$) and Y is a random variable that takes its values in the range of the specification ($rng(R)$), and (X, Y) is in relation R.

- Ord: This specification provides that the final array is ordered, but stipulates no relation to the initial array. In this case, $H(Y|X) = H(Y)$. As we have discussed previously, the entropy of a sorted array of size N whose cells may takes 2^W values is: $H(Y) = N \times W - N \times log(N) = 3200 - 664 = 2536\, bits$.
- Prm: This specification provides that the final array is a permutation of the initial array but stipulates nothing about how array cells are arranged. For an array of size N, this leaves $N!$ possible final array values for each initial array value, hence $H(Y) = \log(N!) = 564\, bits$ for $N = 100$, according to Stirling's approximation.
- $Sort$: Because this specification is deterministic, $H(Y|X) = 0$, since observation of X determines Y.

6.5 Summary

The following Table summarizes our analysis of the sort routine:

7 Concluding Remarks: Assessment and Prospects

At a time when software systems grow increasingly large and complex, it becomes increasingly tenuous / unrealistic to obsess about fault avoidance (developing fault free software) and fault removal (removing faults from developed software). The very least, the goal of fault-free software, by whatever means it is achieved, ought to be combined with the goal of ensuring that the program is adequately equipped to preclude residual fault from causing failure. Whereas traditional software metrics were based on a syntatic analysis of software products, hence reflected such attributes as fault proneness, our semantics based metrics are intended to reflect, not the representation of programs, but their functional properties, not least their ability to avoid failure once faults have caused errors.

Metric	Value	Unit
State Redundancy	$[2435..3115]$	bits
Functional Redundancy	14.62	dimensionless
Non Injectivity	564	bits
Non Determinacy, Ord	2536	bits
Non Determinacy, Prm	564	bits
Non Determinacy, $Sort$	0	bits

Figure 3: Semantic Metrics for a Sorting Routine

7.1 Premises

Our semantic software metrics are defined on the basis of the following premises:

- If software metrics are to give us some indication on the quality (reliability/ dependability/ trustworthiness) of software products, they should be focused on failures rather than faults, for a number of reasons:
  - First, it makes more sense to focus on observable effects (failures) rather than on hypothesized causes (faults).
  - Second, the concept of a fault is not clearly defined: whereas failure is defined with respect to a well-defined reference, namely the product's specification, a fault is defined with respect to an implicit idea of what we think the programmer had meant to write, or should have meant to write. If we consider the program we used in section 2.3, we had assumed that the fault in the program was the statement `s=2*s;`, which should have been `s = s*s;`. Yet it is possible, though not natural, to argue that the faulty statement is not `s=2*s;` but rather `s = s mod 6;`, which should be changed into `s = ((s*s)/4) mod 6`. Again, it is posible, though not natural, to argue that actually both statement are faulty, and we need to replace the first statement by `s = 3*s` and the second statement by `s = ((s*s)/9) mod 6`. Hence strictly speaking, neither the number nor the location of the faults is unique, for the same failure.
  - Several empirical studies show that the correlation between fault dentity and reliability (as measured by Mean Time To Failure) is fairly weak [9]; a program may be reliable and have many faults, and may be unreliable with relatively fewer faults.
- To the extent that functional software qualities (such as correctness, reliability, fault proneness, dependability, etc) pertain to whether the software product behaves according to its specification, the answer to this question cannot be found in the software product alone but must also consider the specification. Our metrics suite includes a measure that reflects the extent to which a specification is flexible in its requirements; we envision to elaborate on this by seeking a measure that quantifies how the non-injectivity of program functions and the non-determinacy of specifications combine to support a broad scope for maskability.
- At the same time as we favor to focus on failure avoidance rather than fault removal, we also favor focusing on a macro-level view of the semantic properties of the software product, rather on minute syntactic details.
- Rather than deriving syntactic metrics then using empirical analysis to discover and validate hypothesized correlations with quality attributes, we favor deriving semantic metrics whose relationship to quality attributes we analyze a priori. This does not dispense us from performing empirical analysis, but we envision the empirical study as supporting the analytical study, rather than substituting for it.
- We readily acknowledge that by focusing on semantic rather than syntactic metrics, we are missing out on an important attribute of software artifacts, namely product complexity, which is often equated with error proneness; but we argue that this is a normal tradeoff, given our focus on failure avoidance rather than fault removal.

Metric	Definition	Interpretation	Quantifies	Application/ Use
State Redundancy: program states	$\kappa(\sigma) = H(S) - H(\sigma)$ S: declared states σ: actual states	Redundancy in state representation	Bandwidth of state-checking Assertions	Error Detection
State Redundancy: programs	$\kappa(\sigma) = H(S) - H(\sigma_F)$ S: declared states σ_F: final state	Redundancy in final state representation	Bandwidth of program-wide Assertions	Failure Detection
Functional Redundancy	$\phi(g) = \frac{H(S)-H(Y)}{H(Y)}$ S: declared state space Y: Random var/ range of g	Redundancy of Program Function	Degree of Functional Duplication	Error Detection and Correction
Non-Injectivity	$\theta(g) = H(X\|G(X))$, G: function of g X: random var / domain of G	Non-Injectivity of program function	Bandwidth of maskable errors	Error Maskability
Non-determinacy	$\chi(R) = H(Y\|X)$ R: specification X: random var / domain of R Y: random var / range of R	Latitude afforded by specification	Bandwidth of Error Tolerance	Error Recoverability

Figure 4: Semantic Metrics: Definitions and Interpretations

7.2 Metrics

In keeping with the foregoing premises, we have derived four semantic metrics, which measure a program's ability to detect errors at run-time and avoid failure.

- A measure of state redundancy, which quantifies the non-surjetivity of state representations, is expressed in Shannon bits, and indicates the bandwidth of assertions that can be checked to ensure state consistency.
- A measure of functional redundancy, which quantifies the non-surjectivity of program functions, is expressed as an abstract number, and indicates the ratio or multiplicity of the program function that can be checked for correctness.
- A measure of maskability, which quantifies the non-injectivity of program functions, is expressed in Shannon bits, and indicates the bandwidth of error that may arise in the program state and still be masked by the program.
- A measure of recoverability, which quantifies the non-determinacy of program specifications, is expressed in Shannon bits, and indicates the bandwidth of loss that a program state can sustain while still satisfying the specification.

Together, these four metrics ought to give the analyst some indication regarding the program's ability to tolerate faults and avoid failure.

7.3 Assessment

It is too early to make a judgement on the road-worthiness of our suite of metrics, other than to say that we have derived them analytically, in a goal-oriented manner, in such a way that, together, they produce a comprehensive picture of a program's fault tolerance potential.

7.4 Prospects

We envision a number of extensions of the current work, most notably:

- Analytical Validation, whereby we envision to derive logical or statistical models that correlate our metrics with the system's reliability, or other quality attributes.

- Empirical Validation, whereby we explore correlations between functional quality attributes (reliability, fault tolerance) and semantic metrics.
- Automated Support, whereby we develop automated support to the derivation of semantic metrics from an analysis of the source code and (structured forms of) the requirements specifications.
- Theoretical Foundations, whereby we explore the concept of bandwidth of an assertion, and its relationship to the entropy of the state defined by the variables that are involved in the assertion.

References

[1] Jacob A Abraham and Dan P. Siewiorek. An algorithm for the accurate reliability evaluation of triple modular redundancy networks. *IEEE Transactions on Computers*, C-23(7):682–692, 1974.

[2] Alain Abran. *Software Metrics and Software Metrology*. John Wiley and Sons, 2012.

[3] J Bansyia, C. Davis, and L. Etzkorn. An entropy based complexity measure for object oriented designs. *Theory and Practice of Object Systems*, 5(2):1–9, 1999.

[4] N. Boudriga, A. Mili, and R. Zalila. An automated tool for specification validation: Design and preliminary implementation. In *Proceedings, Hawaii International Conference on System Sciences*, pages 74–82, Kauai, HI, January 1992.

[5] Chris Brink, Wolfram Kahl, and Gunther Schmidt. *Relational mathematics in Computer Science*. Advances in Computer Science. Springer Verlag, Berlin, Germany, 1997.

[6] Imre Csiszar and Janos Koerner. *Information Theory: Coding Theorems for Discrete Memoryless Systems*. Cambridge University Press, 2011.

[7] Christof Ebert and Reiner Dumke. *Software Measurement: Establish, Extract, Evaluate, Execute*. Springer Verlag, 2007.

[8] Letha H. Etzkorn and Sampson Gholston. A semantic entropy metric. *Journal of Software Maintenance Evolution: research and Practice*, 14:293–310, 2002.

[9] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Company, 1997.

[10] C S Gall, S Lukins, L Etzkorn, L Gholston, P Farrington, D Utley, J. Fortune, and S. Virani. Semantic software metrics computed from natural language design specifications. *IET Software*, 2(1), February 2008.

[11] M.H. Halstead. *Elements of Software Science*. North Holland, Amsterdam, 1977.

[12] E.C.R. Hehner. Quantifying redundancy. Private Correspondence, 2003.

[13] J.C. Laprie. Dependability —its attributes, impairments and means. In *Predictably Dependable Computing Systems*, pages 1–19. Springer Verlag, 1995.

[14] Y. Mashiko and V.R. Basili. Using the gqm paradigm to investigate influential factors for software process improvement. *Journal of Systems and Software*, 36:17–32, 1997.

[15] A. Mili, S. Aharon, and Ch. Nadkarni. Mathematics for reasoning about loop. *Science of Computer Programming*, pages 989–1020, 2009.

[16] Larry Morell and Branson Murill. Semantic metrics through error flow analysis. *Journal of Systems and Software*, 20(3):207–216, March 1993.

[17] Larry J Morell and Jeffrey M Voas. A framework for defining semantic metrics. *Journal of Systems and Software*, 20(3):245–251, March 1993.

[18] Olfa Mraihi, Asma Louhichi, Lamia Labed Jilani, Jules Desharnais, and Ali Mili. Invariant assertions, invariant relations, and invariant functions. *Science of Computer Programming*, (http://dx.doi.org/10.1016/j.scico.2012.05.006), 2012.

[19] Linda Northrop, Peter Feiler, Richard P. Gabriel, John Goodenough, Rick Linger, Tom Longstaff, Rick Kazman, Mark Klein, Douglas Schmidt, Kevin Sullivan, and Kurt Wallnau. *Ultra large Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, July 2006.

[20] Dave Patterson and Armando Fox. Recovery oriented computing— an overview. Technical report, University of California at Berkeley, http://roc.cs.berkeley.edu/ roc_overview.html, 2005.

[21] Jeffrey M Voas and Keith Miller. Semantic metrics for software testability. *Journal of Systems and Software*, 20(3):207–216, March 1993.