To appear ACS/IEEE AICCSA-06 Conference to be held in Dubai/Sharjah during March 8-11, 2006 (http://www.cs.utk.edu/aiccsa06/).

Modeling Redundancy: Quantitative and Qualitative Models

A. Mili, Lan Wu
College of Computer Science
New Jersey Institute of Technology
Newark NJ 07102-1982
mili@cis.njit.edu

F.T. Sheldon
U.S. DOE Oak Ridge National Lab
PO Box 2008, MS 6085, 1 Bethel Valley Rd
Oak Ridge TN 37831-6085
sheldonft@ornl.gov

M. Shereshevsky
Lane Department of EE and CS
West Virginia University
Morgantown WV 26506
m_shereshevsky@hotmail.com

J. Desharnais
Département d'Informatique et de Génie Logiciel
Université Laval
Québec QC G1K 7P4 Canada
Jules.Desharnais@ift.ulaval.ca

August 31, 2005

Abstract

Redundancy is a system property that generally refers to duplication of state information or system function. While redundancy is usually investigated in the context of fault tolerance, one can argue that it is in fact an intrinsic feature of a system that can be analyzed on its own without reference to fault tolerance. Redundancy may arise by design, generally to support fault tolerance, or as a natural byproduct of design, and is usually unexploited. In this paper, we tentatively explore observable forms of redundancy, as well as mathematical models that capture them.

Keywords

Redundancy, Quantifying Redundancy, Qualifying Redundancy, Error Detection, Error Recovery, Fault Tolerance, Fault Tolerant Design, Redundancy as a Feature of State Representation, Redundancy as a Feature of System Function.

1 Redundancy, an Evasive Concept

As a concept, redundancy is widely known and widely referenced; also, at an intuitive level, it is widely (though perhaps cursorily) understood. We have felt the need to model system redundancy in the midst of a research project that we conducted on analyzing a fault tolerance flight control system [1, 4, 5, 6, 8, 9]. Subsequent work led to some tentative, preliminary, insights [2, 7, 15, 16]. What strikes us most about this concept is a paradoxical combination of three premises, which are:

- Even though redundancy is a widely used and widely referenced concept, there appears to be little effort to model it in a generic manner that captures all observable forms.
- In addition to *artificially built-in* redundancy, such as modular redundancy, most systems have ample *natural* redundancy. This natural redundancy is seldom acknowledged and exploited, for example to build fault tolerance and improve system reliability.
- Although it is tantalizingly easy to understand at an intuitive level, redundancy has proven rather hard to model formally in a way that is general and meaningful.

In this paper, we attempt to catalog diverse manifestations of redundancy, then we explore means to model them. In Section 2 we present the many faces of redundancy, that we have observed and attempted to model; then we discuss some questions that we wish to address in the long. In Section 3 we introduce qualitative models of redundancy that we have tentatively explored, and briefly comment on the insights that these models afford us; for the sake of readability, we will keep the discussion informal, referring the interested reader to bibliographic references for technical details. In section 4 we explore a quantification of some aspects of redundancy, and outline issues that

pertain to the quantification of other, more advanced concepts. In the conclusion, we summarize our main results and discuss future prospects.

2 Guises of Redundancy

While redundancy is widely recognized as an important system attribute, it has not been modeled and analyzed in a commensurate manner. Most bibliographic references appear to use this term in a literary sense, rather than a technical, well-defined, widely agreed upon, sense. In this Section we will survey and analyze some of the uses of this term, to form a background for our discussion.

2.1 Forms of Redundancy

Many authors have recognized distinct forms of system redundancy, and have classified redundancy under several categories. In [23] Pullum distinguishes between three classes of redundancy in software systems: *Software Redundancy*, referring to multiplicity of functions; *Information or Data Redundancy*, referring to redundant data representations; *Temporal Redundancy*, referring to redundancy between consecutive values of a given function over time. In [25] Shooman distinguishes between two levels of redundancy, including *Modular Redundancy* (at the module/ component/ subsystem level) and *Micro-code Level Redundancy*; he also distinguishes between two architectures of redundancy, *Parallel Redundancy* (redundant modules running concurrently) and *Standby Redundancy* (a spare module activating in case the main module fails). In [2], Ammar et al distinguish between three classes of redundancy: *Temporal Redundancy* (duplicating functions); *Spatial Redundancy* (duplicating components), and *Informational Redundancy* (duplicating data). In [11] Johnson distinguishes between two forms of hardware redundancy: *Passive Redundancy* (modular redundancy with voting) and *Active Redundancy* (standby sparing). He also distinguishes between three forms of software redundancy: *Information Redundancy* (duplicating data), *Time Redundancy* (check-pointing, re-computation) and *Software Redundancy* (assertion checking, N-version programming). Arora and Kulkarni's *Theory of Fault Tolerance Components* [3] can be viewed as capturing a concept of redundancy, though that is not the author's explicit goal. It differs from our approach by the the way in which it models program execution (a sequence of state transitions, as opposed to an input/ output mapping).

Even though these sources do not acknowledge each other, they recognize the same important distinctions, and give them similar names. From a modeling standpoint, we have found the following categorization to be useful for our purposes:

- *State Redundancy*. This arises when the representation of the system state allows a wider range of values than are needed to represent the set of possible states. This is the traditional form of redundancy, that arises in parity-bit schemes, error correcting codes, modular redundancy schemes, etc.
- *Functional Redundancy*. This form arises when, for example, we compute the same function using three different algorithms, and we take a vote on the outputs. We do not distinguish, in functional redundancy, between whether the components that compute the same function are running concurrently, or in sequence.
- *Temporal Redundancy*. Consider the state defined by two variables: the altitude (Z) and the vertical speed (V_Z) of an aircraft. There is no redundancy between the values of Z and V_Z at a given time t (i.e. $V(t)$ and $Z_V(t)$ can take arbitrary values, for a given t), but there is redundancy between the values of these variables within small time intervals, e.g.

$$Z' = Z + V_Z \times dt.$$

- *Control Redundancy* (for lack of a better name). Control redundancy arises in control applications whenever we can achieve the same effect on the system by several distinct control settings. Our interest in redundancy stems from a project we conducted to analyze a fault tolerant flight control system [1, 5, 8, 9]. This system is fault tolerant in the sense that it can keep flying the aircraft (under some restrictive conditions) even if some control surfaces are lost or if some controls become in-operational. This stems from redundancy between the controls that the system operates: though the throttle, elevators, ailerons, flaps, and rudder have distinct functions, some may be used to make up for the loss of others. In at least two recent accidents of civil aviation (Alaska Airlines 261, January 2000; and US Airways 427, September 1994) investigators believe that despite losing flight surfaces, the flight could *in theory* have been saved [19]. Constructive proof is given by an incident at DFW in 1996 in which a flight was saved despite a malfunction of the flaps [20]; the pilot used the left aileron to compensate for the loss.

This classification is neither complete nor orthogonal; all we can claim for it (even then, subject to further investigation) is that it reflects what we view as distinct forms

of redundancy. It illustrates the diversity of forms of redundancy, and the interest in trying to model them and possibly unify them.

2.2 Questions on Redundancy

The study of redundancy, for all its interest, is not a mere intellectual exercise. We view it as a way to better analyze/ understand complex systems. Examples of questions that we envision, in the long term, include:

- *Modeling Redundancy*. Can we define/ model redundancy as an intrinsic property, independent of fault tolerance capability? What other attributes of a system does redundancy affect? How can we model redundancy in a way that reflects its relevant/ useful properties?
- *Classifying Redundancy*. Is the classification proposed above complete? If not, what other forms need to be considered? Is the classification proposed above orthogonal? If not, how can different categories be unified?
- *Quantifying Redundancy*. Can we quantify redundancy? Can we link the quantity of redundancy to specific attributes, such as fault tolerance capabilities?
- *Applying Redundancy*. Can we use our insights on redundancy to exploit/ use implicit forms of redundancy? For example, in N-version programming, it is conceivable that the volume of redundant information is much greater than $N-1$, since each version has natural/ implicit redundancy; our challenge is to model this excess redundancy and determine whether it can also be combined to support further fault tolerance capability than the scheme of N-version programming makes provisions for (e.g. make the system fault tolerant in the presence of a larger number of faults).

These are broad long term questions, distinct from the preliminary issues that we discuss in this paper.

3 Qualitative Models of Redundancy

Whereas in Section 2 we cataloged forms of redundancy from an external standpoint, by characterizing their observable manifestations, we focus in this Section on models of redundancy, which attempt to provide a mathematical rationale for the observed manifestations.

3.1 Redundancy as a Feature of State Representation

To complement the quantitative model presented above, we discuss in this Section a qualitative model, which equates redundancy with functional attributes of the representation of the system state. In this view, we study redundancy as a representational issue, i.e. as a feature of the relation that maps states to their representations, which is the *state representation relation*. 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 [10].
- 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).

A simple example that illustrates why redundancy can naturally be equated with non-surjective representation relations is the parity bit representation of information in a

computer. Out of eight bits in a byte, seven are used to represent different values, and one is used to represent the parity bit. This makes the representation non surjective since the range of the representation function has a cardinality of (at most) 2^7 while the set of representations has a cardinality of 2^8. This non-surjectivity is crucial to error detection; one out of two representations is illegitimate, making it possible to detect some errors. Had the representation been surjective, an error would map a legitimate state into another legitimate (though incorrect) state, making it impossible to detect any error.

3.2 Redundancy as a Feature of System Function

Whereas the previous Section focuses on the redundancy of state representations, this Section focuses on the redundancy of system functions. Specifically, in this Section we will equate redundancy with fault tolerance capability, and explore how functional attributes of systems and their specifications introduce sources of redundancy that can be used to support fault tolerance. For the sake of readability, we resolve to keep the discussion non-technical, referring the reader to publications [7, 16] for technical details.

3.2.1 Redundancy and Surjectivity

Building on earlier work [14], and on widely accepted fault tolerance ideas [12], we have proposed in [7, 15, 16] a hierarchy of correctness levels for intermediate states in computations:

- *Strict Correctness*, characterizing an error-free state.
- *Maskability*, characterizing a state which, while it may not be strictly correct, will spontaneously avoid failure, nevertheless.
- *Recoverability*, characterizing a state which, while it may not be maskable, does nevertheless contain all the necessary information that allows a recovery function to produce a maskable state.
- *Partial Recoverability*, characterizing a state which, while it may not be recoverable, does nevertheless contain some information about maskable states.
- *Non Recoverability*, which characterizes an erroneous state that contains no information that could be used for recovery.

This hierarchy is highlighted in Figure 1. Before we discuss the link between redundancy and surjectivity, we briefly mention that a relation R from X to Y is said to be *surjective* if and only if its range is all of Y. In [16] we have established the link between this hierarchy and increasing levels of redundancy, in the following terms (see Figure 2):

- Each level of correctness can be characterized as the image set of initial set s_0 by some relation, say C (for strict correctness), M (for maskability), V (for recoverability) and T (for partial recoverability).
- The relations that correspond to these consecutive levels of correctness (C, M, V, T) have increasingly larger ranges, and for each argument, increasingly larger image sets.
- The ability to detect errors, assess damage, perform complete recovery or perform partial recovery represent increasing levels of fault tolerant capability.
- The ability to provide these increasing levels of fault tolerant capability is dependent, according to [16], on relations C, M, V and T being non-surjective. For example, the only way we can detect errors is for C not to be surjective; for if C were surjective, then any value that C produces would look legitimate, and we could not distinguish between correct states and erroneous states —whereas if C is not surjective and produces a state outside its range, we would know for sure that this state is not correct.
- Because C, M, V and T have increasingly larger ranges, it takes an increasingly larger space to make them non-surjective (i.e. to be larger than their ranges). The space of the computation is enlarged by defining additional state variables, which is the source of increased redundancy.

Hence we have established the link between function surjectivity and system redundancy by highlighting two relations: increasing levels of fault tolerance depend on increasing levels of non-surjectivity; increasing levels of non-surjectivity depend on enlarging state spaces, by introducing state variables, which increase redundancy.

3.2.2 Redundancy and Injectivity: Past Functions

A function is said to be *injective* if it maps distinct arguments into distinct images. By and large, we refer to the equivalence relation ($P(x) = P(y)$) that a function P defines on its domain as the *nucleus* of P (represented, in relational notation, by $P\widehat{P}$), and we refer to its equivalence classes as the *level sets* of P (terminology of Mills et al [13, 17, 18]). A function is injective if its nucleus

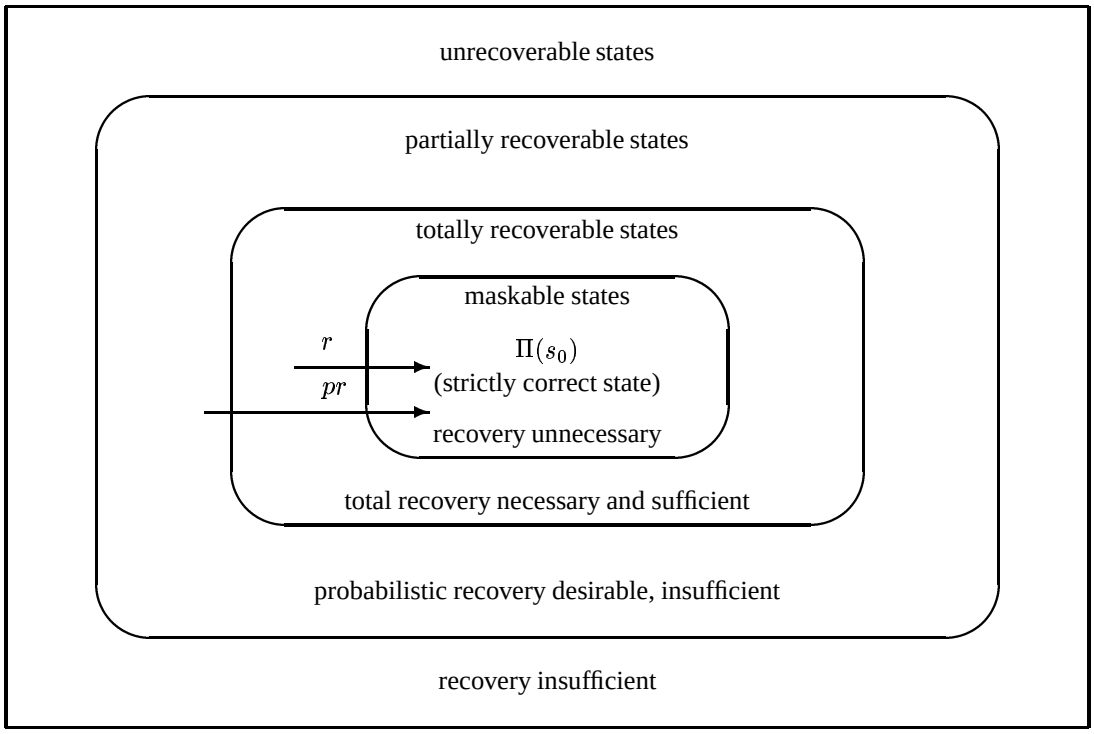


Figure 1: A Hierarchy of Correctness Levels.

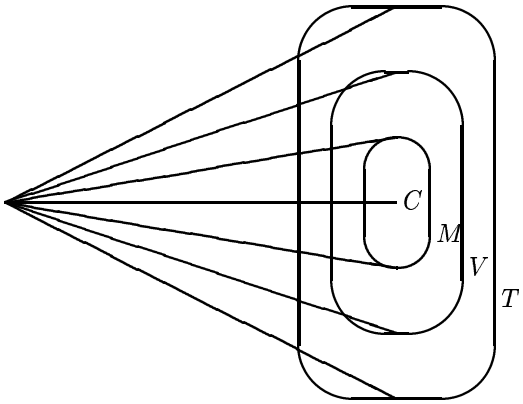


Figure 2: Redundancy as Surjectivity of Relations with Increasing Ranges

is the identity relation (whence its levels sets are singletons), and it grows increasingly less injective as its level sets grow larger and larger. The least injective function is a constant function, which maps all its arguments into the same image; the nucleus of such a function is the total relation (L), which has one equivalence class (the whole space).

To illustrate the relationship between redundancy and injectivity, we show how maskability, recoverability, partial recoverability and un-recoverability can be determined by the degree of injectivity of function P. For the purposes of this discussion, we will introduce this relationship by means of a simple/ simplistic example. We consider the space S defined by an integer variable x, and we consider the following simple program

```
P; K: F
```

where K is a label, and `P` (past) and `F` (future) are defined as follows:

```
P:  x = x mod 6;
F:  x = x mod 9 + 12;
```

We will denote by P and F the functions defined by programs `P` and `F`; whenever no ambiguity arises, we may

confuse a program with the function it computes. If the computation starts with initial state x_0, then at label K we must have state $(x_0 \bmod 6)$. This is the only correct state at label K.

If the past function is incorrect, and instead of computing $(P(x) = x \bmod 6)$ it computes

$$P_1(x) = x \bmod 6 + 18$$

then the states that P_1 produces are not correct, but they are still maskable, in the sense that application of the future function (which takes the mod by 9) after P_1 will cancel out the error produced by P_1 (which mistakenly adds 18 to the correct result).

If the past function is incorrect, and instead of computing $(P(x) = x \bmod 6)$ it computes

$$P_2(x) = x \bmod 12$$

then the states that P_2 computes are not even maskable, but they are recoverable, in the following sense: If we know what is $(x \bmod 12)$, we can derive $(x \bmod 6)$. We say that P_2 *preserves recoverability* with respect to the expected past function P. It is possible to recover from errors caused by P_2 by simply applying $(\bmod 6)$ to the current (potentially erroneous) state.

If the past function is incorrect, and instead of computing $(P(x) = x \bmod 6)$ it computes

$$P_3(x) = x \bmod 3$$

then the states that P_3 computes are not even recoverable, but they are *partially recoverable*, in the following sense: If we know what is $(x \bmod 3)$, we may not know exactly what is $(x \bmod 6)$, but we know something about it. For example, if $(x \bmod 3) = 1$, we know that $(x \bmod 6)$ is either 1 or 4. We then say that P_3 *preserves partial recoverability* with respect to the expected past function $[P]$. We can envision a *probabilistic recovery routine* which preserves the current state or adds 3 to it, and has a 0.5 probability of retrieving the correct state.

If the past function is incorrect, and instead of computing $(P(x) = x \bmod 6)$ it computes

$$P_4(x) = x \bmod 7$$

then the states that P_4 produces are not recoverable, in the following sense: knowing $(x \bmod 7)$ gives us no information whatsoever on the value of $(x \bmod 6)$.

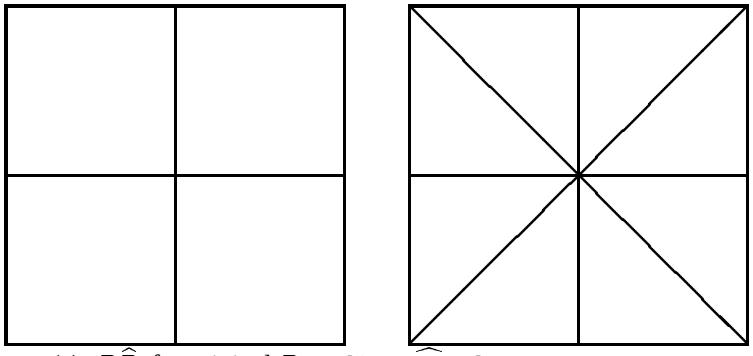
A superficial, intuitive look at this example seems to indicate that a function P' preserves recoverability with respect to an ideal function P if the level sets of P' (i.e. the equivalence classes of the domain of P' modulo the relation $P'(s) = P'(s')$) refine (in the sense of: define a finer partition) the level sets of P. Note, interestingly, that this relation does not involve how P' maps inputs to outputs, as that is a correctness preservation consideration, not a recoverability preservation consideration. What is important, from the standpoint of recoverability preservation, is not what values P' assigns to each level set (if that were wrong, the recovery routine can always correct it), but rather how P' partitions its domain into level sets (as that reflects whether P' maintains sufficient information to compute the correct final result, which is the essence of recoverability preservation). For all these cases except the last, it is possible to recover from errors, using exclusively the current state, with perhaps less than 1.0 probability of successful recovery.

Figure 3 illustrates the hierarchy between the various properties that we have discussed:

- Figure (a) represents the partition of the domain of P by the equivalence relation $P\widehat{P}$. This is identical to the partition of the domain of P by $P_1\widehat{P_1}$.
- Figure (b) represents the partition of the domain of P by the equivalence relation $P_2\widehat{P_2}$. This function preserves recoverability, because if we know what partition we are in by $P_2\widehat{P_2}$, we know what partition we are in by $P\widehat{P}$.
- Figure (c) represents the partition of the domain of P by relation $P_3\widehat{P_3}$. This function preserves partial recoverability, because if we know what partition we are in by $P_3\widehat{P_3}$, we can infer a limited set of possible partitions by $P\widehat{P}$.
- Figure (d) represents the partition of the domain of P by relation $P_4\widehat{P_4}$. This function does not preserve recoverability, because knowing what partition we are in by $P_4\widehat{P_4}$ gives us no indication on what partition by P we are in; the two partitions are perfectly orthogonal.

3.2.3 Redundancy and Injectivity: Future Functions

In the previous Section, we have observed, on a sample example, how we can equate redundancy with the injectivity of past functions. We briefly argue, in this Section, how we can also equate redundancy with the *non*-injectivity of future functions. To illustrate this idea, we again equate redundancy with fault tolerance capability, and consider the following sample example, which is a variation of the example discussed above. We consider the space S defined by an integer variable x, and we consider a program structured as the sequence



(a): $P\widehat{P}$, for original P (b): $P_2\widehat{P_2}$, where P_2 preserves recoverability

(c): $P_3\widehat{P_3}$, where P_3 preserves partial recoverability (d): $P_4\widehat{P_4}$, where P_4 does not preserve recoverability

Figure 3: Degrees of Recoverability

P; K: F.

If, instead of

$$F(x) = x \bmod 9 + 12,$$

the future function computed

$$F_1(x) = x \bmod 3 + 12,$$

(which is less injective than F), then the past function P_2, which was deemed to preserve recoverability, would now be maskable, since

$$(x \bmod 12) \bmod 3 + 12 = (x \bmod 6) \bmod 3 + 12.$$

Likewise, the past function P_3, which was deemed to preserve partial recoverability, would now preserve recoverability, since all we need to know to preserve recoverability with respect to the new future function is

$$x \bmod 3,$$

which is what P_3 computes. Hence when we made the future function less injective (from $x \bmod 6 + 12$ to $x \bmod 3 + 12$) past function P_2 went from preserving recoverability to being maskable and function P_3 went from preserving partial recoverability to preserving recoverability.

Beyond the technicalities of this example, it is easy to see why redundancy increases when past functions grow more injective and future functions grow less injective:

- The injectivity of past functions ensures the preservation of important information.
- The non-injectivity of future functions reduces the amount of information that must be preserved to avoid failure.

3.3 Redundancy as a Feature of System Specifications

If we equate redundancy with fault tolerance, which we have throughout Section 3.2, we must recognize that the non-determinacy of system specifications is an important source of (often unexploited) redundancy. In all the discussions we had so far, we have assumed that for a given input to the system, there exists a single correct output, and have equated recovery with the ability to retrieve this single correct output from current information. But there is usually a wide gap of determinacy between system functions and system specifications [22], for a variety of reasons (design decisions that characterize specific implementations but bear no relation to the system requirements; weak functional requirements, that require approximate values; state variables whose final value does not matter to the user; etc). Taking into account the non-determinacy of specifications would add several levels to the classification depicted in Figure 1; it would in fact double the number of categories, creating an orthogonal classification to that of Figure 1. We do not show the full classification obtained if we were to introduce non-deterministic specifications, but use Figure 4 to illustrate the impact of non-determinacy on maskability; in this Figure, we now have two levels of maskability, one for the system function (F-Maskable) and one for the system specification (S-maskable). Imagine a version of Figure 1 where each level of correctness is duplicated: one level pertains to the system function, and one pertains to the system specification.

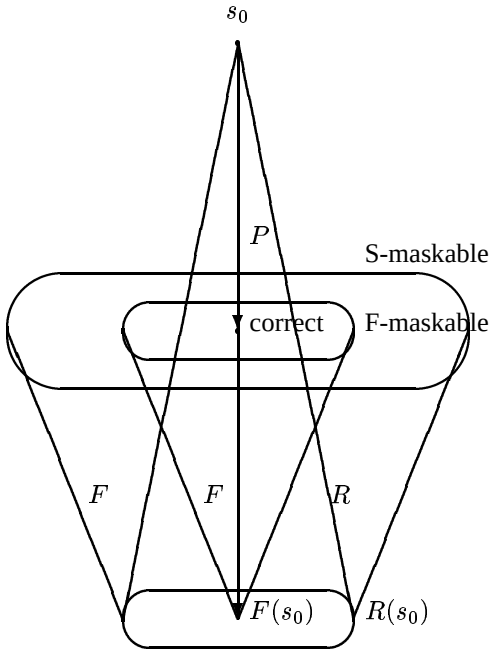


Figure 4: Non Determinacy of Specifications: An additional Dimension of Redundancy

What makes this redundancy model of particular interest to us is our hypothesis (yet to be confirmed) that this is an adequate model for control redundancy, introduced in Section 2. To illustrate this idea, we consider the *Flight Control Loop* depicted in Figure 5, taken from [15]. In this loop, the *Flight Control Software* (FCS) takes as input sensor readings (depicting relevant flight parameters), Pilot Commands (as Auto-Pilot settings) and possibly Navigation Signals (for navigating an approach to a

landing), and produces as output actuator settings. The specification of FCS describes the relation that must be maintained between inputs and outputs. The control redundancy means that the same state of the aircraft can be achieved by more than one control setting; this is reflected by the non-determinacy of the specification of FCS, whereby the same input will be mapped onto a wide range of possible settings, corresponding to equivalent control combinations. This is obviously a tentative interpretation, which we envision to investigate and formalize further in this project.

Another incentive for us to study specification non-determinacy as a possible model of redundancy is that system safety can be modeled as system correctness, but with respect to a weaker specification [15, 21]. Whence we have a hierarchy of three specifications, which are ordered by refinement, and which can be used as references against which correctness, maskability and recoverability can be judged. In [15] we have shown, very cursorily, how reliability and safety concerns can be combined, using a two dimensional matrix of correctness levels (with respect to the correctness requirements, and with respect to safety requirements). We envision to pursue this matter further.

4 A Quantitative Model: Measuring Excess Information

4.1 Space Redundancy

We focus our attention in this Section on state redundancy, and we try to quantify the amount of excess information of the state representation by means of a numeric function that matches the information carried by a state against the number of bits used to represent the state. We envision our function to satisfy the following identities: It takes its values in the set of non-negative real numbers; it takes value zero whenever the state carries no redundancy; it takes value 1 if a redundancy-free state is duplicated; generally, it takes value $N - 1$ whenever a redundancy-free state is duplicated N times. In [24], C. Shannon introduces a quantitative measure of state redundancy as follows:

> "The ratio of the entropy of a source to the maximum value it could have while still restricted to the same symbols will be called its relative entropy. One minus the relative entropy is the redundancy".

We consider a state s ranging over a state space S, we let $P(s)$ be the probability of occurrence of s, and we denote by $H(S)$ the entropy of S modulo probability P and by $M(S)$ the maximal entropy of space S (typically, $M(S) = \log(|S|)$). According to Shannon, the relative entropy of S is given by the following formula

$$\mathcal{E}(S) = \frac{H(S)}{M(S)}.$$

We use Shannon's formula to obtain the *relative* redundancy of space S (though Shannon merely calls it redundancy)

$$\mathcal{R}(S) = 1 - \mathcal{E}(S).$$

As defined by this formula, relative redundancy varies between 0 and 1, and measures the quantity of excess information normalized to the maximum entropy. The redundancy function ($\mathcal{D}$) we are interested in ranges between 0 and infinity, and measures the amount of excess information normalized to the entropy of the space we are representing. We define it by the following formula,

$$\mathcal{D}(S) = \frac{M(S) - H(S)}{H(S)},$$

where the numerator represents excess information and the denominator normalizes the excess information to the actual information being represented. We find readily that the relative redundancy and the (absolute) redundancy satisfy the following equations:

$$\mathcal{D}(S) = \frac{1}{\mathcal{R}(S)} - 1,$$

$$\mathcal{R}(S) = \frac{1}{1 + \mathcal{D}(S)},$$

which are consistent with their respective interpretations.

In practice, we did encounter a difficulty, however: we were not sure how to interpret Shannon's concept of maximal entropy, $M(S)$. Also, we found situations where all possible interpretations of Shannon's definition lead to unsatisfactory results. Examples include:

- If we consider 8 states represented by taking binary codes 000 to 111 and duplicating them (to obtain: 000000, 001001, 010010, 011011, 100100, ... 111111), we are not sure what is the maximum entropy in this case: Is it $log(8) = 3$ since we have only eight values to represent, or is it $log(64) = 6$ since we can represent up to 64 values?

- In the case of Huffman prefix codes, where the length of the code varies according to the probability of occurrence of each symbol, we are not sure what maximal entropy means. There seems to be no way to reflect the property that we have a variable length code,

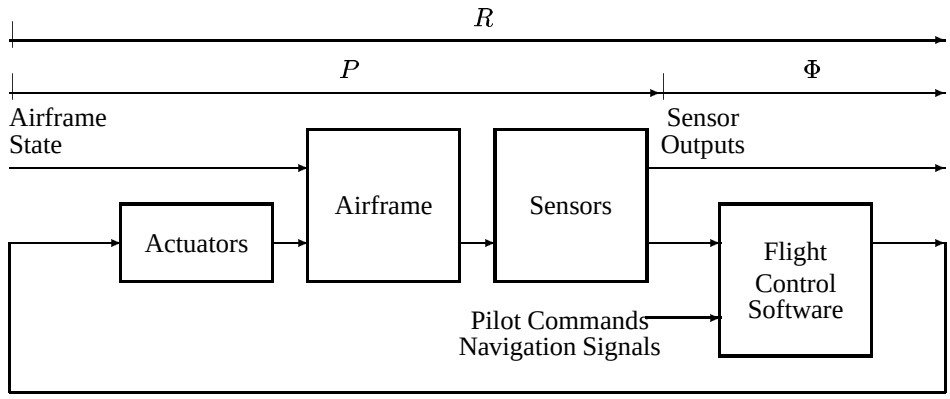


Figure 5: Outline of a Flight Control Loop.

since the maximal entropy is achieved by a uniform probability distribution, which yields a code of constant length.

To overcome this dilemma, we had to reinterpret/ redefine Shannon's formula, by replacing the *maximal entropy* ($M(S)$)with the *weighted length* of the code, which we denote by $W(S)$. Whence we revisit the definition of redundancy and rewrite as:

$$\mathcal{D} = \frac{W(S)}{H(S)} - 1,$$

where $W(S) = \sum_{s \in S} p(s) \times W(s)$, and $p(s)$ is the probability of occurrence of s, and $W(s)$ is the length of s. Intuitively, it makes sense that the formula of redundancy should mention the size of the representation somewhere, which it now does.

To illustrate this function, we consider some sample examples below; in the interest of brevity, we will not show details of our computations, but all we are doing is applying the formula above.

1. **No Redundancy**. If S contains 8 states that are equally likely to occur and are coded on 3 bits, then $\mathcal{D}(S) = 0$.

2. bf Duplication. If S contains 8 states that are equally likely to occur and are coded on 6 bits where the code of each state is duplicated, then $\mathcal{D}(S) = 1$. This value means that one hundred percent of the information needed to represent these states is added to the representation, which reflects the situation at hand.

3. **Error Correcting Code**. We consider a space of 8 values (of equal probability) and we represent it by four bits, say three bits of data and a parity bit. We find that the redundancy function is then equal to: 0.333, whose interpretation is obvious.

4. **Non Uniform Distribution**. If S contains 8 states that are not equally likely to occur and are coded on 3 bits, then $\mathcal{D}(S) > 0$. For example, if the probability distribution is:

   $$(0.03, 0.05, 0.1, 0.12, 0.13, 0.15, 0.2, 0.22)$$

   then the redundancy function yields the value: 0.069.

5. **Tame Distribution**. If the variance in the probability is more tame, the redundancy is much smaller. Hence for the following probability distribution,

   $$(0.09, 0.1, 0.1, 0.1, 0.125, 0.125, 0.14, 0.22)$$

   we find the value: 0.0215.

6. **Huffman Coding**. If we consider the Huffman prefix code obtained from the first probability distribution, i.e.

   $$(0.03, 0.05, 0.1, 0.12, 0.13, 0.15, 0.2, 0.22)$$

   we find the redundancy as: 0.011, which is significantly less than the value found above for fixed size coding (item 4).

7. **Huffman Coding, Tame Distribution**. If we consider the Huffman prefix code obtained from the second probability distribution, i.e.

   $$(0.09, 0.1, 0.1, 0.1, 0.125, 0.125, 0.14, 0.22)$$

we find the redundancy as: 0.012, which is significantly less than the value found above for fixed size coding (item 4) but more than the previous example (item 6), on account of having a tamer probability distribution.

8. **Prefix Code, Non Uniform Distribution**. We consider the non uniform probability distribution,

$$(0.03, 0.05, 0.1, 0.12, 0.13, 0.15, 0.2, 0.22)$$

but this time we use a prefix code that is not necessarily a Huffman code (some codes are longer than they have to be). In this case we find the redundancy value of 0.021, which is greater than the value found for the same probability distribution when we use Huffman coding (item 6).

9. **Non Prefix Code**. If we use the following distribution

$$(0.09, 0.1, 0.1, 0.1, 0.125, 0.125, 0.14, 0.22)$$

and build a non prefix code (by building the huffman tree then placing one symbol on an internal node of the tree), we find the following value of redundancy: -0.081.

The results computed by this formula are consistent with our intuition, for the examples we have explored above. Although we do not consider these developments to be a proof, they illustrate the following premises:

- Redundancy takes a minimal value for Huffman coding.
- The redundancy of Huffman coding decreases as the distribution grows less uniform.
- Redundancy takes non negative values for all lossless codings.
- Redundancy takes negative values for codings that produce a loss of information.

4.2 Functional Redundancy

Whereas in the previous (sub) section we explored means to quantify space redundancy, in this section we briefly touch upon the issue of quantifying functional redundancy. To be sure, the mathematical literature on Shannon's entropy does include definitions of functional redundancy, which equate the redundancy of a function F from X to Y to the conditional entropy $H(X|F(X))$, and seem to measure the non-injectivity of a function. But our discussion of qualitative models of redundancy leads us to the following conclusions:

- We must generalize the concept of redundancy to encompass relations, not just functions.
- What aspect of a relation / function we quantify depends on whether the relation is a past function, a future function, a representation function, or a specification.

This matter is currently under investigation. The first step in this direction consists in exploring a unified model for redundancy that integrates all the relevant factors (the past function, the future function, the specification, the representation function), and reflects the observations we made in the qualitative analysis (that redundancy increases with the injectivity and non surjectivity of past functions, with the non injectivity of future functions, with the non determinacy of specifications, and with the non surjectivity of representation functions).

5 Conclusion: Summary and Prospects

5.1 Summary

In this paper we have attempted to analyze system redundancy from a variety of angles, focusing first on external manifestations of redundancy, then on possible mathematical models that capture it.

Figure 6 synthesizes our analysis by illustrating all the properties that we found to enhance redundancy. Surjectivity is represented by an arrow that points to a circle (symbolizing that the range of the function is smaller than its space); injectivity is represented by parallel arrows (symbolizing the property that different antecedents are mapped onto different images); non-injectivity is represented by arrows that converge to a common target (symbolizing the property that different antecedents are mapped onto a common image); and non-determinacy is represented by a cone (mapping a single antecedent onto several images).

5.2 Prospects

We have offered no definite solutions in this paper, only tentative observations about the multiple forms, multiple models, and multiple observable manifestations of redundancy. We are exploring quantitative and qualitative models that allow us to capture the many dimensions of redundancy that we have discussed in this paper, with the aim of unifying them. We are also exploring means to use the insights derived from these models to make better use of

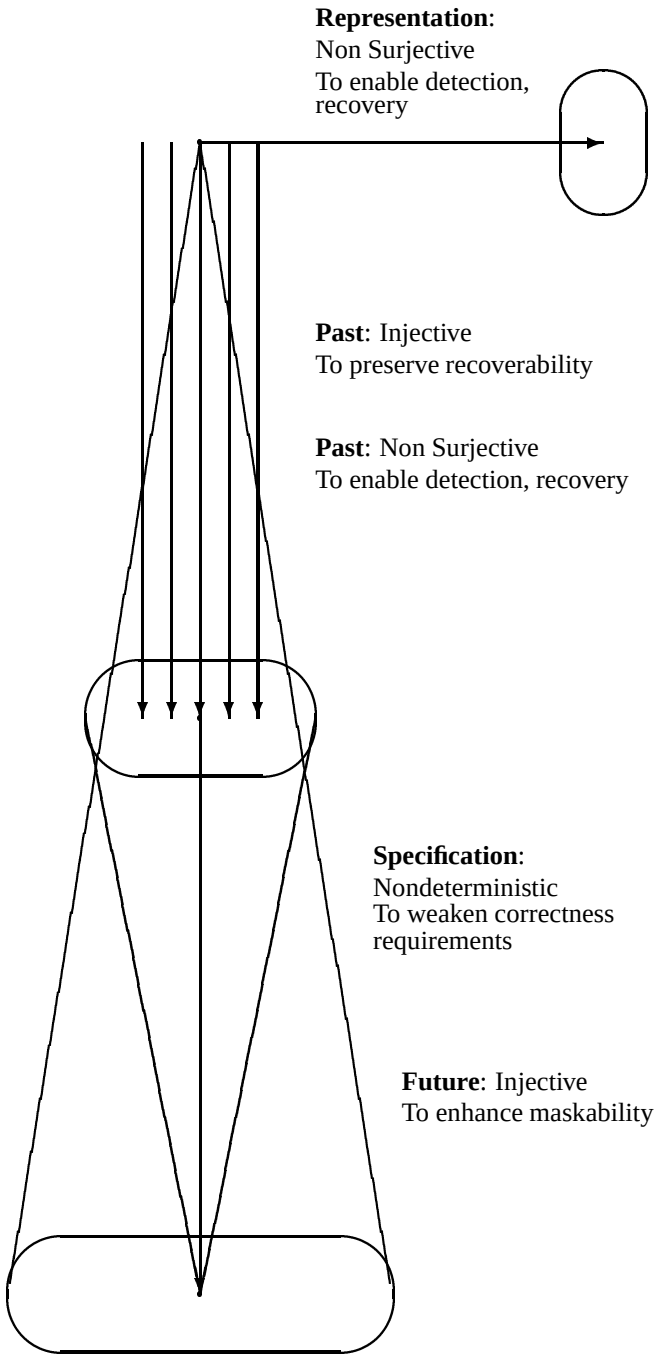


Figure 6: A Synthesis of Redundancy Properties

redundancy, for the purpose of fault tolerance for example.

References

[1] Ch. Alexander, D. DelGobbo, V. Cortellessa, A. Mili, and M. Napolitano. Modeling the fault tolerant capability of a flight control system: An exercise in SCR specifications. In *Proceedings, Langley Formal Methods Conference*, Hampton, VA, June 2000.

[2] H. Ammar, B. Cukic, C. Fuhrman, and Mili. A comparative analysis of hardware and software fault tolerance: Impact on software reliability engineering. *Annals of Software Engineering*, 10, 2000.

[3] Anish Arora and Sandeep S. Kulkarni. Detectors and correctors: A theory of fault tolerance components. In *Proceedings, International Conference on Distributed Computing Systems*, pages 436–443, May 1998.

[4] M. A. Boyd, J. Schumann, G. Brat, D. Giannakopoulou, B. Cukic, and A. Mili. Ifcs project: Validation and verification (v&v) process guide for software and neural nets. Technical report, NASA Ames Research Center, September 2001.

[5] V Cortellessa, A Mili, B Cukic, D Del Gobbo, M Napolitano, and M Shereshevsky. Certifying adaptive flight control software. In *Proceedings, ISACC 2000: The Software Risk Management Conference*, Reston, Va, September 2000.

[6] B. Cukic, A. Mili, and M. Napolitano. Final report: A certification study of an adaptive fault tolerant flight control system. Technical report, West Virginia University, August 2000.

[7] Diego DelGobbo, Mark Shereshevsky, Vittorio Cortellessa, Jules Desharnais, and Ali Mili. A relational characterization of system fault tolerance. *Science of Computer Programming, to appear*, 2005.

[8] D. Del Gobbo and A. Mili. An application of relational algebra: Specification of a fault tolerant flight control system. In *Proceedings, Relational Methods in Software*, Genoa, Italy, April 2001.

[9] D. Del Gobbo and A. Mili. Re-engineering fault tolerant requirements: A case study in specifying fault tolerant flight control systems. In *Proceedings, Fifth IEEE International Symposium on Requirements Engineering*, pages 236–247, Royal York Hotel, Toronto, Canada, 2001.

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

[11] Barry W Johnson. *Design and Analysis of Fault Tolerant Digital Systems*. Addison Wesley, 1989.

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

[13] R.C. Linger, H.D. Mills, and B.I. Witt. *Structured Programming*. Addison Wesley, 1979.

[14] A. Mili. *An Introduction to Program Fault Tolerance: A Structured Programming Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[15] Ali Mili, Frederick Sheldon, Fatma Mili, and Jules Desharnais. Recoverability preservation: A measure of last resort. In *Proceedings, First Conference on the Principles of Software Engineering*, Buenos Aires, Argentina, 2004.

[16] Ali Mili, Frederick T Sheldon, Fatma Mili, Mark Shereshevsky, and Jules Desharnais. Perspectives on redundancy: Applications to software certification. In *Proceedings, 38th Hawaii International Conference on System Sciences*, Hilton Waikoloa Village, Big Islland, HI, 2005.

[17] H.D. Mills, V.R. Basili, J.D. Gannon, and D.R. Hamlet. *Structured Programming: A Mathematical Approach*. Allyn and Bacon, Boston, Ma, 1986.

[18] H.D. Mills, R.C. Linger, and A.R. Hevner. *Principles of Information Systems Analysis and Design*. Academic Press, 1985.

[19] NTSB. NTSB final report identification: DCA94MA076. Technical report, National Transportation Safety Board, 1994.

[20] NTSB. NTSB final report identification: NYC96IA169. Technical report, National Transportation Safety Board, 1996.

[21] David Parnas. System safety. Private Correspondence, 2004.

[22] David L Parnas. Precise description and specification of software. In Daniel M. Hoffman and David M. Weiss, editors, *Software Fundamentals*, chapter 5. Addison Wesley, 2001.

[23] Laura L Pullum. *Software Fault Tolerance Techniques and Implementation*. Artech House, Norwood, MA, 2001.

[24] C. Shannon. A mathematical theory of communication. *Bell Syst. Tech. Journal*, 27:379–423, 623–656, 1948.

[25] Martin L Shooman. *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis and Design*. John Wiley and Sons, New York, NY, 2002.