IEEE Proc. HICSS-38, Testing and Certification of Trustworthy Systems Minitrack, Big Island, Hawaii, Jan. 3-6, 2005

Perspectives on Redundancy: Applications to Software Certification

A. Mili
CCS, NJIT
Newark NJ 07102-1982
mili@cis.njit.edu

F.T. Sheldon†
PO Box 2008, MS 6085, ORNL
Oak Ridge TN 37831-6085
sheldonft@ornl.gov

F. Mili
SECS, Oakland University
Rochester MI 48309-4401
mili@oakland.edu

M. Shereshevsky
LCSEE, West Virginia University
Morgantown WV 26506
m_shereshevsky@hotmail.com

J. Desharnais
Informatique, Université Laval
Québec PQ G1K 7P4 Canada
Jules.Desharnais@ift.ulaval.ca

August 28, 2004

Abstract

Redundancy is a feature of systems that arises by design or as an accidental byproduct of design, and can be used to detect, diagnose or correct errors that occur in systems operations. While it 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 any fault tolerance capability. In this paper, we submit three alternative views of redundancy, which we propose to analyze to gain a better understanding of redundancy; we also explore means to use this understanding to enhance the design of fault tolerant systems.

Keywords

Redundancy, Quantifying Redundancy, Qualifying Redundancy, Error Detection, Error Recovery, Fault Tolerance, Fault Tolerant Design.

1 Introduction: Exploring Redundancy

Redundancy is a feature of systems that arises by design or as an accidental byproduct of design, and generally pertains to an excess of information in the representation of system states or in the execution of system functions. Our interest in redundancy as a subject stems from a research project we had worked on for NASA Dryden, whose topic was the certification study of a fault tolerant flight control system based on analytical redundancy [1, 8, 14, 15]. As we completed this project, we came away with a number of tentative conclusions pertaining to system redundancy:

- Redundancy takes several different forms, not all of which are adequately modeled, understood, and exploited.
- Systems usually carry a great deal of redundancy, some by design but most by accident, only little of which is ever used for fault tolerance.
- Though there are sophisticated means to use redundancy for fault tolerance, only trivial (and sub-optimal) means are generally used in practice.

The purpose of our research is to build on these tentative conclusions by trying to broaden our understanding of redundancy and using new insights to enhance the design of fault tolerant systems.

In this paper, we present some preliminary ideas on our effort to model/ analyze/ understand redundancy, and discuss our prospects for gaining new insights, and using them to enhance the practice of fault tolerant systems design.

2 Multiple Forms of Redundancy

One of the conclusions we had drawn from our earlier project investigating a fault tolerant flight control system is that redundancy takes multiple forms. It is possible that all these forms may fit under a single generic model, and one of the goals of our discussion is precisely to explore this possibility. For the time being, we can only perceive them as distinct, and wish to present some of them in the sequel, to give the reader some sense of what we mean:

- *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

†This manuscript has been authored by a contractor of the U.S. Government (USG) under DOE Contract # DE-AC05-00OR22725. The USG retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for USG Purposes.

states. This is the traditional form of redundancy, that arises in parity-bit schemes, error detecting codes, 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, to protect against possible faults.

- *Relational Redundancy* (for lack of a better name). Typically, most system functions are non-injective, and most system specifications are non-deterministic. These properties are a great source of redundancy, in the sense that they allow systems to deviate from their intended function while still avoiding failure.

- *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 arbitrarily 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). The flight control system that we analyzed in [1, 8, 14, 15] 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 inoperational. This stems from redundancy between the controls that the system operates: though the throttle, elevators, ailerons 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 [26]. Constructive proof is given by an incident at DFW in 1996 in which a flight was saved despite a malfunction of the flaps [27]; the pilot used the left aileron to compensate for the loss.

This classification is neither complete nor orthogonal; it is meant to illustrate the diversity of forms of redundancy, and the interest in trying to model them and possibly unify them. Also, this list highlights a large untapped potential, that can possibly be exploited to enhance system fault tolerance in a systematic manner; but before it can be tapped, it must first be adequately modeled and understood.

3 Multiple Views of Redundancy

In addition to taking many forms, redundancy can be viewed from multiple perspectives, which we explore below. For the sake of argument, and with some loss of generality, we restrict our discussion in the remainder of this paper to state redundancy. We have identified three complementary views of state redundancy:

- *Redundancy as a Quantitative Measure of Duplication*. It is possible to view redundancy as the duplication of system state information. This duplication may be total (as in modular redundancy) or partial (as in parity bit schemes, for examples). We propose to quantify the amount of redundancy in a state by a numeric function, whose formula we discuss in the sequel.

- *Redundancy as a Qualitative Feature of State Representation*. We consider the *State Representation Relation* as a mapping from state values to state representations, and we characterize redundancy by the algebraic properties of this relation.

- *Redundancy as a Qualitative Measure of Fault Tolerance Capability*. In this view, we equate redundancy with fault tolerance, and we characterize redundancy levels by successive degrees of fault tolerance (such as error detection, damage assessment, backward error recovery, forward error recovery, etc).

In the following section, we analyze in some detail these three views of redundancy, and set the stage for our subsequent discussions.

3.1 Redundancy as Non Surjectivity

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 *ambiguous representation*. Consider the representation of signed integers between -7 and +7 using a sign-magnitude format; zero has two representations, -0 and +0 [17].

- 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).

3.2 Redundancy as Excess Information

Whereas in the previous section we characterized state redundancy by the properties of the state representation function, in this section we measure redundancy by means of a numeric formula which reflects how much excess information the representation of the system state carries. To this effect we consider a state s ranging over a state space S, and we let $W(s)$ be the number of bits that are used to represent s, and $P(s)$ the probability of occurrence of s. The redundancy of S is given by the following formula

$$\delta(S) = \sum_{s \in S} P(s) \times \frac{(W(s) + log(P(s))}{W(s)}.$$

This formula is due to Hehner [17], and can be explained as follows: $-log(P(s))$ is the amount of information carried by s, and $W(s)$ is the size of the actual representation of s; the difference between these two quantities is the redundancy of the representation of s, which we normalize by dividing it by $W(s)$. By taking the prorated sum of all the state redundancies, we obtain the redundancy of the whole state space. To illustrate the meaning of this function, we consider some sample examples:

- If S contains 8 states that are equally likely to occur and are coded on 3 bits, then $\delta(S) = 0$.

- 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 $\delta(S) = 1$. 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.

- *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.

- If S contains 8 states that are not equally likely to occur and are coded on 3 bits, then $\delta(S) > 0$. 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: 0.118. 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: 0.0489.

- Using Huffman coding rather than fixed size coding for uneven probability distributions reduces (or preserves) the redundancy of the state. Hence for the latter distribution above,

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

  we find the following vector of lengths under Huffman coding, $(4, 4, 3, 3, 3, 3, 3, 2)$, which produces a value of redundancy of: 0.0041, which is, expectedly, less than the value found under fixed size coding (0.0489).

3.3 Redundancy as Fault Tolerance Capability

Whereas the previous section quantifies redundancy by assigning it a number, this section attempts to characterize it by the fault tolerance capability that it affords us. For the sake of our discussions, we will borrow results from section 4, where we present detailed mathematics for characterizing fault tolerance capabilities.

In section 4, we define three levels of correctness of a state s in the course of a computation:

- *Strict correctness*, whereby a state is the precise image of the initial state s_0 by the correct past function Π; in Figure 1, this state is represented by the bullet at the center.

- *Maskability*, whereby a state is maskable if and only if subsequent computation will map it into a correct final state for the initial state s_0; in Figure 1, maskable states are represented by the first oval around the center.

- *Recoverability*, whereby a state is recoverable if and only if it carries sufficient information to be transformed (by a recovery routine) into a maskable state;

in Figure 1, recoverable states are represented by the second oval around the center.

It is plain from the (summary) definitions above and from Figure 1 that:

- A correct state is maskable (because we assume that $\Pi \circ \Phi$ refines R, as per section 4).
- A maskable state is recoverable (taking *identity* as the recovery routine).

Traditional sources on system fault tolerance advocate the following life-cycle for providing fault tolerance:

- *Error Detection*, i.e. the task of determining whether the current state is equal to the expected state at the current stage of the computation.
- *Damage Assessment*, i.e. the task of determining whether the state, while being strictly incorrect, is still maskable, or at least recoverable.
- *Error Recovery*, i.e. the task of mapping the current incorrect state into a correct state, provided the current state is recoverable (otherwise recovery is futile) and not maskable (otherwise recovery is unnecessary).

We submit the thesis that successive levels in this hierarchy require successive amounts of redundancy to be performed, in the following sense:

- *Correctness*. According to Definition 6, a state s is correct at cut-point (i.e. stage of the computation) C for initial state s_0 if and only if

  $$(s_0, s) \in \Pi.$$

  At cut-point C, we have access to state s but we do not have access to the initial state s_0; hence we cannot in principle check correctness using the formula above. However, if s had some redundancy, we could check some necessary condition of correctness on s, which, if it does not hold, proves that s is not (strictly) correct. A trivial necessary condition for $(s_0, s) \in \Pi$ that is independent on s_0 is the condition $\exists s_0 : (s_0, s) \in \Pi$, which can be rewritten as $s \in \mathit{rng}(\Pi)$. This is interesting: If Π is surjective, then the condition above is trivial (i.e. takes the constant value $\mathbf{true}$ for all s), and it is not possible to detect whether s is or is not correct. Hence, interestingly, error detection is contingent upon the non-surjectivity of function Π; this is reminiscent of, but different from, the view we have taken in section 3.1, whereby redundancy is equated with the non-surjectivity of the representation function.
- *Maskability*. According to Proposition 2, a state s is maskable if and only if

  $$(s_0, s) \in \kappa(R, \Phi).$$

  A trivial (and most general) sufficient condition of maskability that is independent of s_0 is: $\exists s_0 : (s_0, s) \in \kappa(R, \Phi)$. We re-interpret this condition as: $s \in \mathit{rng}(\kappa(R, \Phi))$. This condition is trivial (i.e. equal to $\mathbf{true}$ for all s) if and only if $\mathit{rng}(\kappa(R, \Phi)) = S$; in that case, it is not possible to detect maskability because all states satisfy this condition. We can detect maskability only as soon as $\mathit{rng}(\kappa(R, \Phi))$ is a strict subset of S, i.e. as soon as $\kappa(R, \Phi)$ is not surjective. From the discussions of section 4, we can establish that

  $$\mathit{rng}(\Pi) \subseteq \mathit{rng}(\kappa(R, \Phi)).$$

  Hence requiring that relation $\kappa(R, \Phi)$ be non-surjective is a stronger condition that requiring than relation Π be non-surjective.
- *Recoverability*. Without inspecting the complex characterization of recoverability (in section 4), let us simply note that the set of recoverable states is larger than the set of maskable states (since all maskable states are recoverable, with trivial recovery routine $r = I$); in order for recoverability to be verifiable, the set of recoverable states must be smaller than S.

In summary, we find that

- Past functions that preserve correctness, maskability, and recoverability have successively larger ranges.
- Successive fault tolerant capabilities are dependent on these functions being non-surjective.
- The larger the range of a function, the more redundancy (i.e. excess representation information) is required to make the space S larger than its range.

We find that increasing levels of fault tolerant capability are dependent on the non-surjectivity of functions with increasingly larger ranges, and that making these functions non-surjective requires increasing levels of redundancy. It is in this sense that redundancy is equated with fault tolerance capability.

4 Background: Mathematics for Fault Tolerance

Due to space limitations, we will limit our presentation in this section to the main definitions and propositions.

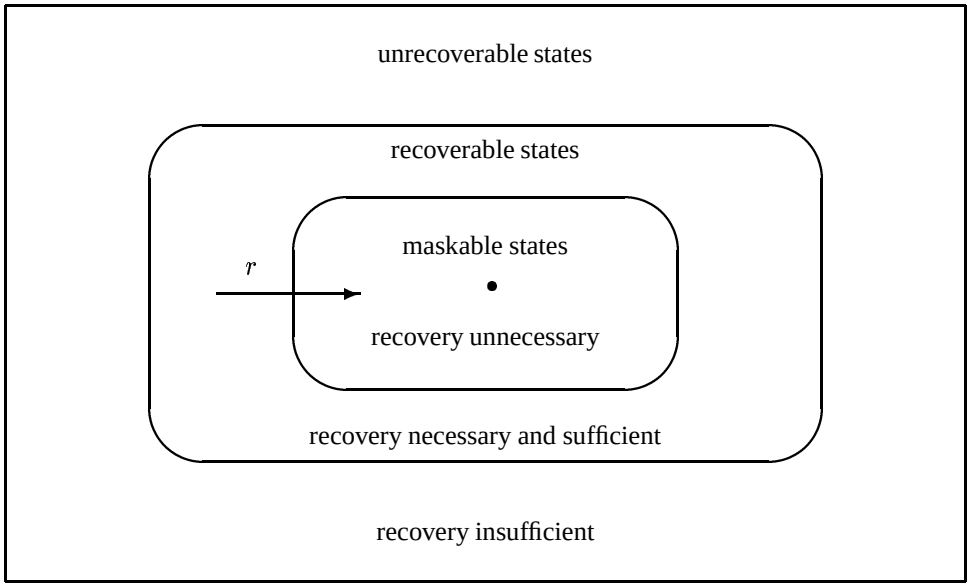

Figure 1: A Hierarchy of Correctness Levels.

The main result of this section is Proposition 3, page 7, which provides a necessary and sufficient condition for a (possibly) faulty component to preserve the recoverability of the system state.

4.1 Elementary Concepts of Relational Mathematics

We represent the functional specification of systems or system parts by relations; without much loss of generality, we consider homogeneous relations, and we denote by S the space on which relations are defined. As a specification, a relation contains all the (input, output) pairs that are considered correct by the specifier. Constant relations include the *universal* relation, denoted by L, the *identity* relation, denoted by I, and the *empty* relation, denoted by ϕ. Because relations are sets, we use the usual set theoretic operations between relations. Operations on relations also include the *inverse*, 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 $R \circ R' = \{(s, s')|\exists t : (s, t) \in R \wedge (t, s') \in R'\}$. The *prerestriction* (resp. *post-restriction*) of relation R to predicate t is the relation $\{(s, s')|t(s) \wedge (s, s') \in R\}$ (resp. $\{(s, s')|(s, s') \in R \wedge t(s')\}$). The *domain* of relation R is defined as $dom(R) = \{s|\exists s' : (s, s') \in R\}$. The *nucleus* of relation R is the relation denoted by $\mu(R)$ and defined by $R\widehat{R}$. We say that R is *deterministic* (or that it is a *function*) if and only if $\widehat{R}R \subseteq I$, and we say that R is *total* if and only if $I \subseteq R\widehat{R}$, or equivalently, $RL = L$. We say that R is *regular* if and only if $R\widehat{R}R \subseteq R$ [25]. We define an ordering relation on relational specifications under the name *refinement ordering*: A relation R is said to *refine* a relation R' if and only if $RL \cap R'L \cap (R \cup R') = R'$. We abbreviate this property by $R \sqsupseteq R'$ or $R' \sqsubseteq R$. We admit without proof that this relation is a partial ordering. We also admit that, modulo traditional definitions of total correctness [10, 16, 23], the following propositions hold.

- A program P is correct with respect to a specification R if and only if $[P] \sqsupseteq R$, where $[P]$ is the function defined by P.
- $R \sqsupseteq R'$ if and only if any program correct with respect to R is correct with respect to R'.

Intuitively, R refines R' if and only if R represents a stronger requirement than R'. In conjunction with the refinement ordering, we introduce a composition-like operator, which we denote by $R \triangleright R'$ and define by $R \triangleright R' = RR' \cap \overline{R\overline{R'L}}$. The main characteristic of this operator, for our purposes, is that unlike traditional composition, it is monotonic with respect to the refinement ordering, i.e. if $R \sqsupseteq Q$ and $R' \sqsupseteq Q'$, then $R \triangleright R' \sqsupseteq Q \triangleright Q'$.

We introduce two related division-like operations on relations, which will play a crucial role in our subsequent discussions. Because the monotonic product is not commutative (nor is the simple product), we need two division-like operators: a right division and a left division.

Definition 1 *The (conjugate)* kernel *of relation R with relation R' is the relation denoted by $\kappa(R, R')$ and defined by*

$$\kappa(R, R') = \overline{\overline{R}\widehat{R'}} \cap L\widehat{R'}.$$

The (conjugate) cokernel *of relation R with relation R' is the relation denoted by $\Gamma(R, R')$ and defined by*

$$\Gamma(R, R') = \kappa(\widehat{R}, (\widehat{RL \cap R'}))\,\widehat{}\,.$$

The kernel is due to [9]; both the kernel and the cokernel are discussed in some detail in [11, 12], where the interested reader is referred. Similar relational operators have been investigated at length [2, 3, 4, 6, 18, 19, 20, 21, 28]. For the purposes of our discussion, the most interesting properties of kernels and cokernels are articulated in the following proposition.

Proposition 1 *The in-equation $R \sqsubseteq X \,\square\, R'$ has a least refined solution in X if and only if $RL \subseteq \kappa(R, R')L$. Under this condition, its solution, which we call the* left residual *of R with respect to R' and denote by $R /\!\!/ R'$, is given by*

$$R /\!\!/ R' = \kappa(R, R').$$

The in-equation $R \sqsubseteq R' \,\square\, X$ has a least refined solution in X if and only if $RL \subseteq R'L \wedge L \subseteq \overline{(\widehat{RL \cap R'})\overline{R}L}$. Under this condition, its solution, which we call the right residual *of R with respect to R' and denote by $R' \backslash\!\!\backslash R$, is given by*

$$R' \backslash\!\!\backslash R = \Gamma(R, R').$$

The first clause of this proposition is due to [9] (proposition 4.5), where a proof is given. The second clause of this proposition is due to [11], where a proof is given.

4.2 Elementary Concepts of Fault Tolerance

4.2.1 Fault, Error and Failure

In [22], Laprie defines *failure*, *error* and *fault* in the following terms:

> A system **failure** occurs when the delivered service deviates from fulfilling the system **function**, the latter being *what the system is intended for*. An **error** is that part of the system state which is *liable to lead to subsequent failure*; an error affecting the service is an indication that a failure occurs or has occurred. The *adjudged or hypothesized cause* of an error is a **fault**.

In this section we briefly present working definitions of fault, error and failure, and we illustrate them on a sample system structure. We consider a space S, defined by a set of *state variables*. We consider a compound function from S to S, and we decompose this function into the (monotonic) product of two functions: a function Π, to which we refer as the *past function*; and a function Φ, to which we refer as the *future function*. We assume that the compound function $(\Pi \,\square\, \Phi)$ is due to satisfy some requirements that are captured in the relational specification R, i.e. $\Pi \,\square\, \Phi \sqsupseteq R$. Furthermore, we suppose that by structuring the system as the product of two components Π and Φ, the designer has specific expectations of what requirements functions Π and Φ must satisfy. In our discussion, we focus on the fault behavior of the past function; to this effect, we distinguish between the *ideal* past function, which we will (continue to) denote by Π, and the *actual* past function, which we will denote by Π'. We have the definition.

Definition 2 *A* fault *is a feature of a system that precludes it from operating according to its specification.*

Specifically, for our purposes, the past function Π' has a fault if and only if

$$RL \cap \Pi \neq RL \cap \Pi'.$$

Whereas *fault* is a feature of a function (the past function), error is a feature of a state (the current state). In order to identify states of interest, we introduce the label/cut-point C, which defines the state of the computation after application of the past function.

Definition 3 *We say that there is an* error *at some cut-point C of a computation if and only if the value of the system state at cut-point C differs from the expected value at that step.*

If we let C be the cut-point that marks the range of the past function and the domain of the future function, then we say that we have observed an error at cut-point C if we find an element s of S that satisfies the following conditions:

$$\exists s_0 : s_0 \in dom(R) \wedge (s_0, s) \in \Pi' \wedge (s_0, s)\neg \in past.$$

Definition 4 *We say that there is a* failure *of a system if and only if the actual output of the system for some input is not a correct output.*

With respect to our system structure, there is a failure for initial state s_0 if and only if

$$s_0 \in dom(R) \wedge (s_0, (\Pi' \,\square\, \Phi)(s_0)) \notin R.$$

4.2.2 Fault Tolerance

The definitions given above allow us to define *fault tolerance*.

Definition 5 *A system is said to be* fault tolerant *if and only if it has provisions for avoiding failure after faults have caused errors.*

In fault tolerance, we resign ourselves to the presence of faults in the system, and we take measures to ensure that faults do not cause failure.

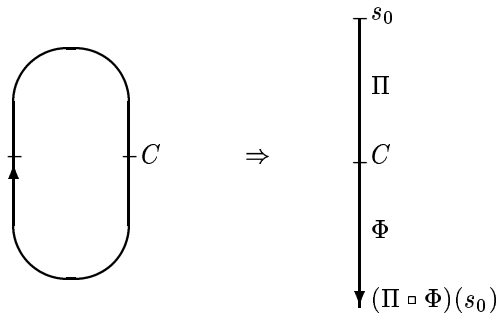

Figure 2: Unwinding a Control Loop.

4.3 Degrees of Error

4.3.1 Imperatives of Error Detection: Condition of Correctness

In the context of the system structure that we discussed in section 4.2.1, we are interested in characterizing the state that we obtain after applying function Π', right before function Φ is applied. In control applications, we often witness the case when control information runs in a closed loop. In such cases, we propose to cut the loop in two cut-points, as we will illustrate in figure 3: the point that will characterize the domain of function Π; and the point that will characterize the domain of function Φ. We let C designate the cut-point (label) where function Π feeds into function Φ, and we let s_0 be an initial state of function Π; see figure 2.

Definition 6 *State s at cut-point C is said to be* correct *for initial state s_0 if and only if*

$$(s_0, s) \in \Pi.$$

If and only if state s is not correct at cut-point C, we say that we are observing an *error* at cut-point C.

4.3.2 Imperatives of Damage Assessment: Condition of Maskability

Definition 7 *A state s is said to be* maskable *at cut-point C for initial state s_0 with respect to R if and only if $(s_0, \Phi(s)) \in R$.*

We have the following proposition, which characterizes maskable states in closed form.

Proposition 2 *A state s is maskable at cut-point C for initial state s_0 with respect to R if and only if*

$$(s_0, s) \in \kappa(R, \Phi).$$

We interpret damage assessment as the process of addressing the following two questions, given that we have a state s which is known (following error detection) to have an error:

- *Whether Recovery is Necessary*, i.e. whether or not the state is maskable: if it is, then recovery is unnecessary.
- *Whether Recovery is Sufficient*, i.e. whether or not the state is recoverable: if it is not recoverable, then recovery is insufficient to ensure failure-freedom.

See Figure 1. In this section we discussed the condition of maskability; in the next section we discuss recoverability.

4.3.3 Imperatives of Error Recovery: Condition of Recoverability

A state is recoverable if and only if it contains all the necessary information to produce a maskable state. In this section, we attempt to give meaning to the concept of recoverability.

Definition 8 *A state s is said to be* recoverable *at cut-point C for initial state s_0 with respect to R if and only if there exists a function, say r, such that $r(s)$ is maskable.*

We resolve to model this property under the form $(s_0, s) \in \pi$, for some function π, and we must now characterize functions π that produce recoverable states. We then anticipate that the condition of recoverability of s be written as the conjunction of two clauses: a clause of the form $V(\pi, \Phi, R)$, which expresses under what condition relation π produces only recoverable states s for each initial state s_0 with respect to Φ and R, and a clause of the form $(s_0, s) \in \pi$, which merely expresses that s is obtained from s_0 by applying a function (or relation) that only produces recoverable states. We focus our attention on the first clause; when a function π satisfies this condition, we say that it *preserves recoverability* with respect to Φ and R. We submit the following definition.

Definition 9 *Given specification R and future function Φ, we say that function π* preserves recoverability *(or is* recoverability preserving*) with respect to future function Φ and specification R if and only if there exists a function r (recovery function) such that*

$$\pi \circ r \sqsupseteq \kappa(R, \Phi).$$

In other words, a past function preserves recoverability if and only if we can combine it with some recovery function r to achieve (or exceed) maskability. Proposition 1 provides a necessary and sufficient condition for the existence of such a function r, which we use to derive the following proposition.

Proposition 3 *Given specification R and future function Φ, a past function π preserves recoverability if and only if*

$$KL \subseteq \pi L \wedge L \subseteq \overline{(\widehat{KL \cap} \pi)\overline{K}L}.$$

where K is an abbreviation for $\kappa(R, \Phi)$.

Proposition 4 *If past function π preserves recoverability with respect to future function Φ and specification R, then*

$$r = \Gamma(\pi, \kappa(R, \Phi))$$

satisfies the equation: $\pi \,\square\, r \sqsupseteq \kappa(R, \Phi)$.

Because the demonic composition is monotonic, we infer from this Proposition that any relation r' that refines r satisfies this condition, a fortiori. Hence r can be used as the specification of recovery routines.

Example. We consider a simplified flight control loop defined by a flight control system and an airframe (along with sensors and actuators), and we decompose / unwind the loop as follows:

- The past function, Π, is the function defined by the mapping from actuator inputs to sensor outputs; see Figure 3.
- The future function, Φ, is the function of the flight control software ($FCSw$), which analyzes sensor outputs and pilot commands, and computes actuator inputs, that are then fed to the actuators.
- The specification R maps actuator inputs and pilot commands into new actuator inputs; we let R capture the minimal requirements that must be satisfied to preserve the safety of the flight.

The condition of recoverability preservation can be interpreted as the minimal requirement that the past function π (implemented by the aggregate *actuators-airframe-sensors*) must satisfy at all times to ensure the survivability of the flight. If this condition is not satisfied, then no recovery is possible, irrespective of what component $FCSw$ may do. □

The following proposition provides a simple (simpler) sufficient condition of recoverability preservation.

Proposition 5 *Given a specification R and a future function Φ, if R is regular and the following conditions are satisfied*

$$R\widehat{\Phi}L \subseteq \pi L \text{ and } \mu(\pi) \subseteq \mu(R)$$

then π preserves recoverability with respect to future function Φ and specification R.

The most important clause of this proposition is the condition

$$\mu(\pi) \subseteq \mu(R)$$

which we illustrate by a simple example. For the sake of simplicity, we assume that both R and π are function, to facilitate the discussions. Under this hypothesis, the condition above provides that in order for π to preserve recoverability with respect to R, it has to define a finer partition of its domain than R. We submit that this condition is actually quite intuitive: the condition is not saying anything about what values function π assigns (since the function can be faulty) but is saying something about the nucleus of the function (i.e. how the function divides its domain into equivalence classes, or equivalently, how much information the function preserves about its argument). Furthermore, the condition is providing that function π preserves recoverability if its nucleus is as fine or finer than that of R. We present a simple example to show how intuitive this is: Imagine that R is the function

$$R = \{(s, s') | s' = s \,\mathbf{mod}\, 6\}.$$

The nucleus of R is then the equivalence relation

$$\mu(R) = \{(s, s') | s \,\mathbf{mod}\, 6 = s' \,\mathbf{mod}\, 6\},$$

which has six equivalence classes (the congruence classes modulo 6). A past function π preserve recoverability with respect to R if its nucleus is a subset of $\mu(R)$, i.e. if it defines a finer partition of its domain than R does. Examples of functions that preserve recoverability include:

$$\pi_1 = \{(s, s') | s' = s \,\mathbf{mod}\, 12\}, \pi_2 = \{(s, s') | s' = s \,\mathbf{mod}\, 24{+}5\},$$

Examples of functions that do not preserve recoverability include:

$$\pi_3 = \{(s, s') | s' = s \,\mathbf{mod}\, 5\}, \pi_4 = \{(s, s') | s' = s \,\mathbf{mod}\, 7\}.$$

The reader can easily see why it is possible to recover from an error produced by the first two functions, but impossible with the last two.

5 Conclusion

In [24] we had advocated the use of a wide range of methods to verify/ validate complex systems, by virtue of the law of diminishing returns, and of our observation that a system can be verified against complementary subspecifications in an additive manner; the approach we advocated is contingent upon the methods being formulated in the same mathematical model. In this paper we have taken a step further by attempting to capture ideas of system fault tolerance using relational mathematics, which have traditionally been used for program proving/ programming language semantics/ programming calculi, etc. [5, 7, 29]. The most important result of this paper, we feel, is Proposition 3, which highlights the condition that must hold between a target (ideal) function that we must

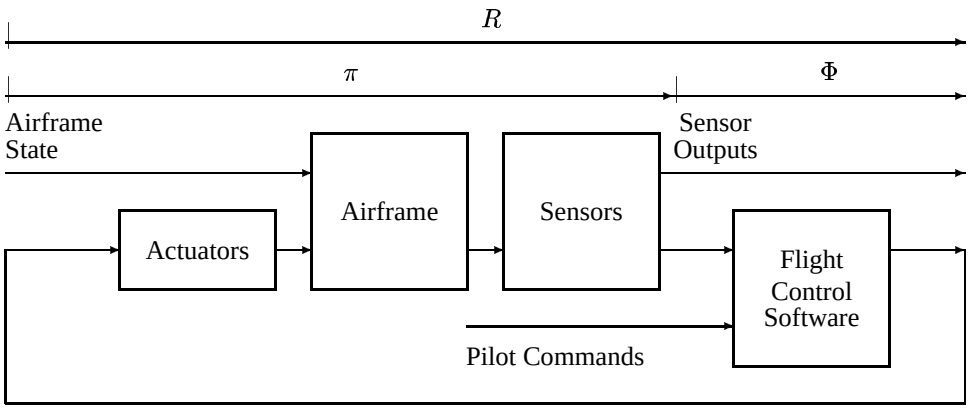

Figure 3: Outline of a Flight Control Loop.

compute, and the minimal requirement that actual (possibly faulty) functions must fulfill to satisfy the recoverability property. In order to simplify the condition and give the reader some intuition for its meaning, we have considered a sufficient condition for recoverability preservation, which provides that an actual function preserves the recoverability of an ideal function if the level sets it defines over its domain define a finer partition than the levels sets of the ideal function —which is intuitively understandable. In its generalized form, Proposition 3 formulates this condition for specifications that are not deterministic, nor even regular.

We have not explored applications and extensions of this work in much detail, though we envision the following applications:

- *Proving Recoverability Preservation as a Substitute for Proving Correctness.* In a complex system, where it may be unrealistic or unreliable to prove that the past function produces only correct (or maskable) states, we may instead want to prove that the past function preserves recoverability and takes measures to recover when needed. Because recoverability preservation is a much weaker property than maskability, the former may be easier and produce more dependable conclusions.

- *Proving Recoverability Preservation as a Complement for Proving Correctness*. Proving maskability / correctness and proving recoverability preservation need not be viewed as mutually exclusive. As we advocated in [24], they can be done simultaneously, though with different component specifications.

- *Using Recoverability Preservation to Catalog Recoverable Faults*. The research discussed in this paper stems from an earlier project whose purpose was to model, specify and analyze a fault tolerant flight control system [8, 15]. The key idea of this system is that it should be able to continue flying an aircraft even after the aircraft has lost some flight surfaces or the control of some flight surfaces or the feedback from some sensors; clearly, this is possible only for a limited amount of damage. We argue that the condition of recoverability preservation can be used to catalog those fault modes that can indeed be recovered from, and eventually, what recovery action must be applied for these fault modes. Some faults are so extensive (e.g. loss of major surfaces, loss of control of major actuators) that there is no way to recover, no matter what the flight control system does. The condition of recoverability preservation allows us to distinguish between faults that can in principle be recovered from (with appropriate provisions in the flight control system) from faults that cannot be recovered from (and the flight control system is not to blame).

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] R. Backhouse, P. DeBruin, G. Malcolm, E. Voermans, and J. Van der Woude. A relational theory of data types. In *Proceedings, Workshop on Constructive Algorithms: The Role of Relations in Program Development*, Hollum Ameland, Holland, September 1990.

[3] R. Berghammer, G. Schmidt, and H. Zierer. Symmetric quotients. Technical Report TUM-I8620, Technische Universitaet Muenchen, Muenchen, Germany, 1986.

[4] R. Berghammer, G. Schmidt, and H. Zierer. Symmetric quotients and domain constructions. *Information Processing Letters*, 33:163–168, 1989.

[5] Rudolf Berghammer and Gunther Schmidt. Relational specifications. In C. Rauszer, editor, *Proc. XXXVIII Banach Center Semester on Algebraic Methods in Logic and their Computer Science Applications*, volume 28 of *Banach*, pages 167–190, Warszawa, 1993. PolishC.

[6] G. Birkhoff. *Lattice Theory*. American Mathematical Society, Providence, RI, 1967.

[7] Ch. Brink, W. Kahl, and G. Schmidt. *Relational Methods in Computer Science*. Springer Verlag, January 1997.

[8] 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.

[9] J. Desharnais, A. Jaoua, F. Mili, N. Boudriga, and A. Mili. A relational division operator: The conjugate kernel. *Theoretical Computer Science*, 114:247–272, 1993.

[10] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[11] M. Frappier. A relational basis for program construction by parts. Technical report, University of Ottawa, October 1995.

[12] M. Frappier, J. Desharnais, and A. Mili. Unifying program construction and modification. *Logic Journal of the International Interest Group in Pure and Applied Logics*, 6(2):317–340, 1998.

[13] D. Del Gobbo and B. Cukic. Validating on-line neural networks. Technical report, Lane Department of Computer Science and Electrical Engineering, West Virginia University, December 2001.

[14] 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.

[15] 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.

[16] D. Gries. *The Science of programming*. Springer Verlag, 1981.

[17] E.C.R. Hehner. Private correspondence on quantifying redundancy. Technical report, University of Toronto, Toronto, Ont, Canada, 2003.

[18] C.A.R. Hoare and et al. Laws of programming. *Communications of the ACM*, 30(8):672–686, 1987.

[19] C.A.R. Hoare and J.F. He. The weakest prespecification. *Fundamentae Informaticae*, IX:Part I: pp 51–58. Part II: pp 217–252, 1986.

[20] B. J´onsson. Varieties of relational algebras. *Algebra Universalis*, 15:273–298, 1982.

[21] M.B. Josephs. An introduction to the theory of specification and refinement. Technical Report RC 12993, IBM Corporation, July 1987.

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

[23] Z. Manna. *A Mathematical Theory of Computation*. McGraw Hill, 1974.

[24] A. Mili, B. Cukic, T. Xia, and R. Ben Ayed. Combining fault avoidance, fault removal and fault tolerance: An integrated model. In *Proceedings, 14th IEEE International Conference on Automated Software Engineering*, pages 137–146, Cocoa Beach, FL, October 1999. IEEE Computer Society.

[25] A. Mili, J. Desharnais, F. Mili, and M. Frappier. *Computer Program Construction*. Oxford University Press, 1994.

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

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

[28] G. Schmidt and T. Stroehlein. *Relationen und Graphen*. Springer-Verlag, Berlin, Germany, 1990.

[29] G. Schmidt and T. Stroehlein. *Relations and Graphs, Discrete Mathematics for Computer Scientists*. EATCS-Monographs on Theoretical Computer Science. Springer Verlag, 1993.