Relational Characterizations of System Fault Tolerance[*]

V. Cortellessa,
Dipartimento di Informatica
Universita dell'Aquila, Via Vetoio
Coppito, L'Aquila, 67010 Italy
cortelle@di.univaq.it

D. Del Gobbo, M. Shereshevsky
Lane Dept of Computer Science and Electr. Eng.
West Virginia University
Morgantown WV 26506-6109, USA
{diego,smark}@csee.wvu.edu

J. Desharnais,
Département d'informatique et de génie logiciel
Université Laval
Québec, QC G1K 7P4, Canada
Jules.Desharnais@ift.ulaval.ca

A. Mili,
College of Computing Science
New Jersey Inst. of Technology
Newark NJ 07102-1982, USA
mili@cis.njit.edu

June 7, 2004

Abstract

Fault tolerance is the ability of a system to continue delivering its services after faults have caused errors. We have argued, in the past, that complex and/or critical systems are best validated by a wide range of methods, including proving, testing, and fault tolerance; we have also argued that in order to use these methods in concert, we need to cast them in a common framework. In this paper, we present mathematical characterizations of fault tolerance properties, using a relational calculus.

Keywords

Programming Calculi, Relational Mathematics, System Fault Tolerance, Fault, Error, Failure.

1 Introduction: Characterizing Fault Tolerance Attributes

Fault tolerance is the ability of a system to avoid failure (i.e., to keep behaving according to specifications) after faults (in the system's design / implementation) have caused errors (i.e., the appearance of incorrect / contaminated / incoherent states). In [22] we have argued that complex and / or critical systems are best validated by deploying a wide range of methods, including testing, proving and fault tolerance, by virtue of the following premises:

- The law of diminishing returns advocates the use of diverse methods, whereby each method is applied wherever / whenever its impact is maximal.
- Complex specifications can be decomposed into simpler specifications in a refinement-compatible manner, so that different methods can be applied to different components.
- Some component specifications lend themselves to some methods more than others, and some methods are more applicable to some component specifications than to others; whence a careful mapping of methods to component specifications can significantly improve overall effectiveness.

[*]This work was funded by the NASA IV&V Laboratory (Independent Verification and Validation) in Fairmont, WV, through a grant in 2000-2001 of the USIP program.

We have also found that in order for the results of different methods applied to different component specifications to be additive, all the methods must be framed within the same mathematical model. In this paper, we take this argument to its logical conclusion by attempting to model fault tolerance properties using a mathematical tool that we, along with other researchers [5, 7, 26], have used to model programming calculi, program proving methods, and programming language semantics; the tool in question is relational mathematics.

In section 2 we briefly introduce the background of this paper by discussing in turn some simplified ideas of fault tolerance and some elementary concepts of relations. Then, in section 3 we present a stepwise characterization of some key fault tolerance properties. The most important of these is the sufficient condition of recoverability preservation (Proposition 3), which provides a minimal condition that a (possibly erroneous) system component must satisfy in order to always produce provably recoverable states. Proposition 6 provides an intuitive understanding of recoverability preservation by giving a simple sufficient condition for recoverability preservation. In section 4 we summarize our findings and discuss some preliminary venues for applications and extensions of our results.

2 Background for System Fault Tolerance

2.1 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 *converse*, denoted by $\widehat{R}$ or $R^\frown$, 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 *range* of relation R is denoted by $rng(R)$ and defined as *dom*$(\widehat{R})$. The *nucleus* of relation R is the relation denoted by $\mu(R)$ and defined by $R\widehat{R}$. For any R, the nucleus of R is symmetric and reflexive on *dom*(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 $R\widehat{R}R \subseteq R$ [23]; this property is known to other authors as *difunctionality* [7]. For a regular relation R, the nucleus of R is transitive, hence defines an equivalence relation on *dom*(R). For a regular relation R, each equivalence class of *dom*(R) modulo $\mu(R)$ contains all the elements that have the same image set by R. Given a relation R on S and an element s in S, we let the *image set* of s by R be denoted by $s.R$ and defined by $s.R = \{s'|(s, s') \in R\}$.

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

$$RL \cap R'L \cap (R \cup R') = R'.$$

In set theoretic terms, this equation means that the domain of R is a superset of (or equal to) the domain of R', and that for elements in the domain of R', the set of images by R is a subset of (or equal to) the set of images by R'. This is similar, of course, to refining a pre/postcondition specification by weakening its precondition and/or strengthening its postcondition [15, 24]. 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, 15, 21], 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.

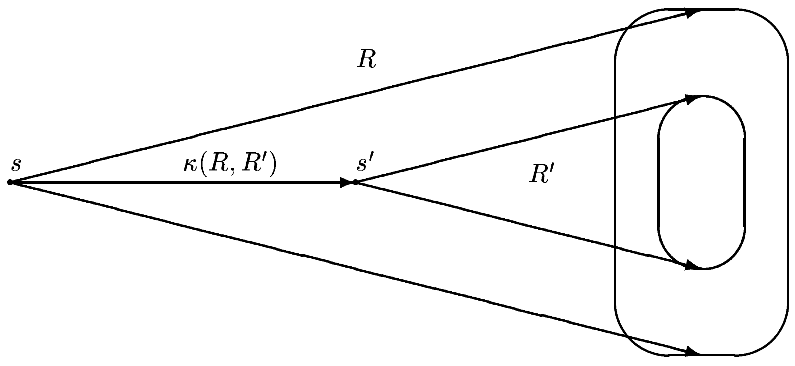

Figure 1: The Conjugate Kernel: $\kappa(R, R')$ as a Solution to: $R \sqsubseteq X \mathbin{\square} R'$.

- $R \sqsupseteq R'$ 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'. We admit without proof that any relation R can be refined by a deterministic relation, i.e. a function. In conjunction with the refinement ordering, we introduce a composition-like operator, which we denote by $R \mathbin{\square} R'$, refer to as the *monotonic product*, and define by

$$R \mathbin{\square} R' = RR' \cap \overline{R\overline{R'}L}.$$

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 $R \mathbin{\square} R' \sqsupseteq Q \mathbin{\square} Q'$.

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*

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

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, 16, 17, 18, 19, 25]. A set theoretic interpretation of the kernel is given in the following formula, and illustrated in Figure 1.

$$\kappa(R, R') = \{(s, s') | \emptyset \neq s'.R' \subseteq s.R\}.$$

For the purposes of our discussion, the most interesting properties of kernels and cokernels are articulated in the following proposition.

Proposition 1 *The inequation $R \sqsubseteq X \mathbin{\square} R'$ 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 inequation $R \sqsubseteq R' \mathbin{\square} X$ has a least refined solution in X if and only if $RL \subseteq R'L \wedge L \subseteq \overline{(R\widehat{L \cap} R')\overline{R}L}$. 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. The interpretation of the kernel as the solution of the equation $R \sqsubseteq X \mathbin{\square} R'$ is clearly visible in Figure 1.

2.2 Elementary Concepts of Fault Tolerance

2.2.1 Fault, Error and Failure

In [20], 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**.

For the sake of our study, and without significant loss of generality, we reinterpret Laprie's definitions in relational/ functional terms.

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*; a state variable may be a natural variable (such as the temperature of a site, the speed of a vehicle) or an artificial variable (such as the voltage that measures the temperature, the odometer reading, or digital versions thereof). 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 $(\Pi \circ \Phi)$ does satisfy some requirements that are captured in the relational specification R, i.e., $\Pi \circ \Phi \sqsupseteq R$. 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

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

Intuitively, one would think that there is a fault in the past function as soon as the actual past function is different from the ideal past function. In fact, this is not necessarily the case: if the difference between Π and Π' pertains to values that are outside the domain of R, then there is no fault.

Whereas *fault* is a feature of a function (specifically, the past function in our sample structure), error is a feature of a state (specifically, the state obtained by applying the past function to some initial state). In order to identify states of interest, we introduce the label / cutpoint 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 cutpoint C of a computation if and only if the value of the system state at cutpoint C differs from the expected value at that step.*

If we let C be the cutpoint 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 cutpoint C if we find an element s of S that satisfies the following conditions:

$$\exists s_0 : s_0 \in \mathit{dom}(R) \wedge (s_0, s) \in \Pi' \wedge (s_0, s) \notin \Pi.$$

In other words,

$$\exists s_0 : (s_0, s) \in RL \cap \Pi' \cap \overline{\Pi}.$$

This can also be formulated as:

$$s \in \mathit{rng}(RL \cap \Pi' \cap \overline{\Pi}).$$

Errors are possible at cutpoint C if and only if

$$\mathit{rng}(RL \cap \Pi' \cap \overline{\Pi}) \neq \emptyset.$$

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

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

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

We transform this condition as follows:

$$\begin{array}{ll}
 & \exists s : s_0 \in dom(R) \wedge (s_0, s) \in (\Pi' \circ \Phi) \wedge (s_0, s) \in \overline{R} \\
\Leftrightarrow & \qquad \{ \text{ interpreting } s_0 \in dom(R) \text{ } \} \\
 & \exists s : (s_0, s) \in RL \cap (\Pi' \circ \Phi) \cap \overline{R}. \\
\Leftrightarrow & \qquad \{ \text{ definition of domain } \} \\
 & s_0 \in dom(RL \cap (\Pi' \circ \Phi) \cap \overline{R}).
\end{array}$$

2.2.2 Fault Tolerance

The definitions given above allow us to define *fault tolerance* [1].

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. Such measures include the following steps:

- *Detecting Errors*. This step involves checking conditions on the current state, possibly in relation with previous states (including the initial state). There are several degrees of correctness that we may want to check, as we discuss subsequently.
- *Assessing Damage*. This step involves assessing the extent of the damage caused by the fault on the state of the computation, and making a decision on what recovery action must be taken.
- *Error Recovery*. This step involves correcting the current state of the computation according to the findings of the damage assessment step, and resuming the computation.
- *Fault Removal*. While the three preceding steps deal with the current computation and are performed on-line for the sake of salvaging the current computation, this step deals with the system itself, and can be performed off-line. It consists of identifying the fault that caused the error and removing it.

The three first steps deal with the error, whereas the last step deals with the fault.

3 Fault Tolerance Properties

In section 2.2, we have listed the following steps as being phases of the fault tolerance process:

- Error detection.
- Damage Assessment.
- Error Recovery.

In order to carry out error detection, we need to characterize the condition under which a state at label C is *correct*, i.e., is the exact image of the initial state by the ideal past function. Also, in order to carry out damage assessment, we need to characterize the condition under which a state at label C is *maskable*, i.e. (irrespective of whether it is correct), application of the future function will produce a failure-free output. Finally, in order to carry out error recovery, we need to characterize the condition under which a state at label C is *recoverable*, i.e., it contains enough information for a recovery routine to map it to a maskable state. These characterizations will be discussed in the sequel.

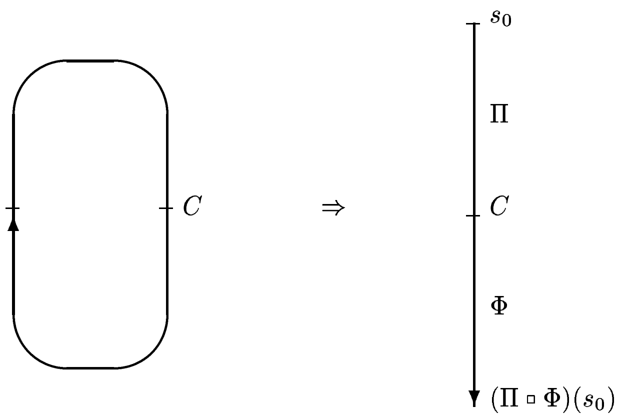

Figure 2: Unwinding a Control Loop.

3.1 Correctness

In the context of the system structure that we discussed in section 2.2.1, 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 cutpoints, as we will illustrate in Figure 4: 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 cutpoint (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 cutpoint C is said to be* correct *for initial state s_0 with respect to past function Π if and only if*

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

If and only if state s is not correct at cutpoint C, we say that we are observing an *error* at cutpoint C. Error detection relies on the condition of correctness to detect errors.

3.2 Maskability

Strict correctness, in the sense of definition 6, is not necessary for failure-freedom; whence the concept of maskability. We further refine the definition of failure (definition 4) by observing that there is not necessarily a single *expected output*, but there may in fact be a large set of correct outputs. We reflect this observation by resolving that failure is defined, not with respect to a deterministic function (namely $\Pi \circ \Phi$) but rather with respect to a potentially non-deterministic relation, which reflects the (minimal) requirements that the system must meet; we denote this relation by R and we assume, a priori, that we have

(1) $\quad \Pi \circ \Phi \sqsupseteq R.$

In other words, under the hypothesis of fault freedom, the system does meet its specification; faults are modeled by deviations in the past function from Π, the *ideal* past function. Because the actual past function, Π', may be faulty, we have no assurance that $\Pi' \circ \Phi$ refines R. We have the following definition.

Definition 7 *A state s is said to be* maskable *at cutpoint C for initial state s_0 and future function Φ with respect to R if and only if*

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

Note that, *by definition*, the condition of maskability is not dependent on function Π (the past function). We have the following proposition, which characterizes maskable states in closed form.

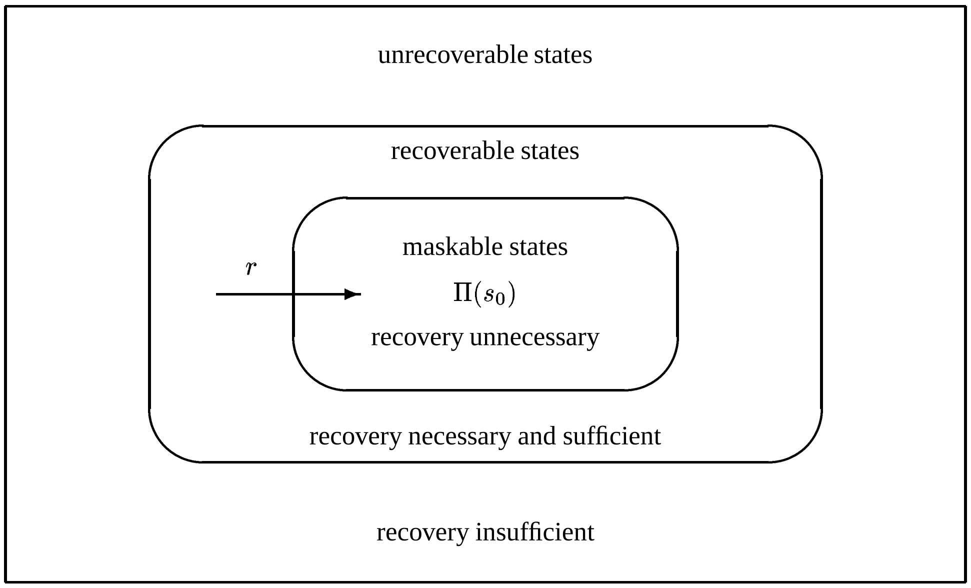

Figure 3: A Hierarchy of Correctness Levels.

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

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

Note that the correctness equation

$$\Pi \circ \Phi \sqsupseteq R.$$

establishes the condition of existence of solutions to the equation

$$X \circ \Phi \sqsupseteq R,$$

since the substitution $X := \Pi$ provides a feasible solution to this equation. From definition 7, we infer that $(s_0, s) \in R\widehat{\Phi}$. Because Φ is a function, we know (from [9]) that $R\widehat{\Phi} = \kappa(R, \Phi)$.

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 3.

3.3 Recoverability

A state is recoverable if and only if it contains all the necessary information to produce a maskable state. It may fail to be maskable itself, but it does have to contain all the required 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 cutpoint C for initial state s_0 and future function Φ with respect to R if and only if there exists a function, say r, such that $r(s)$ is maskable at cutpoint C for state s_0 and function Φ with respect to specification R.*

Implicit in this definition is the requirement that r is not dependent on s, of course: the same function r must recover all states that are recoverable at the given cutpoint. 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 function 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 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. Before we discuss the mathematics of recoverability preservation, we illustrate it with some examples:

- If R is the specification of a sorting routine and Φ is the iteration of a selection sort then relation π would be the specification $Perm$, which merely provides that the array at cutpoint C is a permutation of the initial array. As long as the array is a permutation of its original value, it is possible to produce a final sorted array; if we lose one cell of the array, we no longer have a recoverable state, i.e., we cannot avoid failure, no matter what we do to the array.

- We let the space be the set of natural numbers and R specify that we must compute 5 plus the remainder by 6 of the argument, i.e.,
$$R = \{(s, s')|s' = 5 + (s \,\mathbf{mod}\, 6)\}.$$
Further, we let Π and Φ be defined as
$$\Pi = \{(s, s')|s' = s \,\mathbf{mod}\, 6\},$$
$$\Phi = \{(s, s')|s' = s + 5\}.$$
We submit that all functions of the form
$$\pi = \{(s, s')|s' = s \,\mathbf{mod}\, (6 \times N)\},$$
for all $N \geq 1$, preserve recoverability. Indeed, if we know the remainder of s_0 by 18, say, we can always derive its remainder by 6; but if all we know about s_0 is its remainder by 5, or by 9, for example, then we cannot possibly extract from it the remainder of s_0 by 6.

To reflect this intuition, 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 1 provides the existence of a *relation*, say ρ, which satisfies the equation

$$\pi \Box \rho \sqsupseteq \kappa(R, \Phi).$$

Given that any relation can be refined by a function, we let r be a function that refines ρ. The monotonicity of the $\Box$ operator and the transitivity of the refinement ordering lead us to the result of proposition 3. The interest of this proposition (by contrast with the definition) is that it highlights the fact that recoverability preservation is a tripartite property that involves only the past function π, the future function Φ and the specification R —and does not involve the recovery routine r. If a past function π does preserve recoverability, we must worry about what recovery function to apply. The following proposition gives the least refined (i.e., the optimal, in effect) specification for the recovery routine.

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

Because the monotonic product is monotonic with respect to refinement, we infer from this proposition that any relation r' that refines r satisfies the equation $\pi' \Box r \sqsupseteq \kappa(R, \Phi)$, a fortiori. Hence r can be used as the (minimal) specification of recovery routines, which map recoverable states into maskable states.

Before we discuss sufficient conditions of recoverability preservation, we consider a brief example.

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 of the aggregate made up of the airframe and the sensors and actuators that are attached to it. This function maps the current state of the aircraft and current actuator inputs into a new state of the aircraft, represented by the sensor outputs, as shown in Figure 4.
- The future function, Φ, is the function of the flight control software (FCS), which analyzes the state of aircraft (represented by sensor outputs) and the pilot commands, and computes the actuator inputs (that are then fed to the actuators).
- The specification R represents a relation we wish to impose between the current state of the aircraft and the pilot commands on one hand, and the new state of the aircraft on the other hand. Specification R can be used, for example, to enforce a minimal safety requirement that must be preserved at all time to ensure 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. On the other hand, the specification of a recovery routine, given by Proposition 4, represents the minimal requirement that must be satisfied by a recovery routine that must be invoked prior to FCS whenever we suspect an error. According to Proposition 4, application of this recovery routine prior to FCS ensures that we satisfy the safety requirement R even in the absence of an error that results from a fault in the past function.

Under normal (fault-free) operating conditions, the aggregate of actuators, airframe and sensors delivers function Π. But under fault prone conditions, this aggregate may produce a different function, say π. In [13] we discuss how we can derive the specification of a behavioral envelope which captures all the possible functions defined by π under a variety of pre-cataloged fault modes. What Proposition 3 provides is the minimal requirement that π must satisfy (refine) to ensure recoverability; so long as π refines this minimal requirement, FCS can, theoretically, apply a corrective action to recover. This condition can also be used to test fault hypotheses: a fault mode for which π does not refine the minimal requirements should not be supported, because it cannot be recovered from. $\Box$

The following propositions provide sufficient conditions of recoverability preservation. Before we consider the propositions, we introduce a lemma that will be needed in their proof. The following relational identities (where f is a function) are used in these proofs.

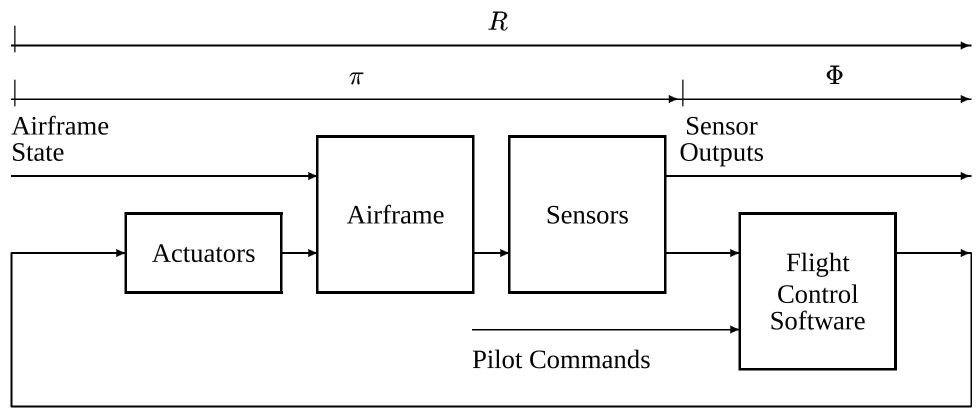

Figure 4: Outline of a Flight Control Loop.

$$
\begin{array}{ll}
(1) & \widehat{f}f \subseteq I \;\Rightarrow\; f\overline{R} \subseteq \overline{fR} \\
(2) & \widehat{f}f \subseteq I \;\Rightarrow\; f\overline{R} = fL \cap \overline{fR} \\
(3) & PQ \subseteq R \Leftrightarrow \widehat{P}\overline{R} \subseteq \overline{Q} \Leftrightarrow \overline{R}\widehat{Q} \subseteq \overline{P}
\end{array}
$$

Lemma 1 *Given a function f and an arbitrary relation Q such that $f\widehat{f}Q \subseteq Q$ and $fL \subseteq QL$. Then $L \subseteq \overline{\widehat{f}\,\overline{Q}}L$.*

Proof.

$$
\begin{array}{ll}
 & L \subseteq \overline{\widehat{f}\,\overline{Q}}L \\
\Leftrightarrow & \qquad \{ \text{ Boolean algebra } \} \\
 & \widehat{f}L \cup \overline{\widehat{f}L} \subseteq \overline{\widehat{f}\,\overline{Q}}L \\
\Leftrightarrow & \qquad \{ \widehat{f}\,\overline{Q} \subseteq \widehat{f}L\text{; Boolean algebra; because } f \text{ is a function, } f\widehat{f}f = f\text{, hence } \widehat{f}f\widehat{f} = \widehat{f} \; \} \\
 & \widehat{f}f\widehat{f}L \subseteq \overline{\widehat{f}\,\overline{Q}}L \\
\Leftrightarrow & \qquad \{ \text{ (3) } \} \\
 & f\overline{\overline{\widehat{f}\,\overline{Q}}L} \subseteq \overline{f\widehat{f}L} \\
\Leftrightarrow & \qquad \{ \text{ Complementation } \} \\
 & f\widehat{f}L \subseteq \overline{f\overline{\overline{\widehat{f}\,\overline{Q}}L}} \\
\Leftarrow & \qquad \{ \text{ (1) and } f\widehat{f}L = fL \; \} \\
 & fL \subseteq f\overline{\widehat{f}\,\overline{Q}}L \\
\Leftrightarrow & \qquad \{ \text{ (2) } \} \\
 & fL \subseteq fL \cap \overline{f\widehat{f}\,\overline{Q}}L \\
\Leftarrow & \qquad \{ \text{ By (3), the assumption } f\widehat{f}Q \subseteq Q \text{ is equivalent to } f\widehat{f}\,\overline{Q} \subseteq \overline{Q}\text{; Boolean algebra } \} \\
 & fL \subseteq QL
\end{array}
$$

qed

Using this lemma, we introduce the following proposition, which provides a sufficient condition of recoverability preservation.

Proposition 5 *If $R\widehat{\Phi}L \subseteq \pi L$ and $(R\widehat{\Phi}L \cap \pi)(R\widehat{\Phi}L \cap \pi)\,\widehat{}\,R\widehat{\Phi} \subseteq R\widehat{\Phi}$, then π preserves recoverability with respect to future function Φ and specification R.*

Proof. Let us transform the condition of existence of a solution r to $\pi \circ r \sqsupseteq \kappa(R, \Phi)$, i.e., the condition for recoverability, as given in Proposition 3.

$$
\begin{array}{ll}
 & KL \subseteq \pi L \ \wedge \ L \subseteq \overline{(KL \cap \pi)\,\hat{}\ \overline{K}}\,L \\
\Leftrightarrow & \qquad\qquad \{ \text{ Definition of } K \text{ and } \Phi \text{ is a function } \} \\
 & R\widehat{\Phi}L \subseteq \pi L \ \wedge \ L \subseteq \overline{(R\widehat{\Phi}L \cap \pi)\,\hat{}\ \overline{R\widehat{\Phi}}}\,L
\end{array}
$$

Thus, the condition $R\widehat{\Phi}L \subseteq \pi L$ is necessary (note that it might be violated by taking $\pi = \phi$). Assume that it holds. It will be used in the proof of $L \subseteq \overline{(R\widehat{\Phi}L \cap \pi)\,\hat{}\ \overline{R\widehat{\Phi}}}\,L$. The latter condition follows quite directly from Lemma 1 by instantiating f and Q as follows:

$$
\begin{array}{l}
f := R\widehat{\Phi}L \cap \pi \\
Q := R\widehat{\Phi}
\end{array}
$$

The relation f is a function, since π is. The assumption $f\widehat{f}Q \subseteq Q$ of Lemma 1 is satisfied because it is a hypothesis of our proposition. Using the assumption $R\widehat{\Phi}L \subseteq \pi L$, we show that the assumption $fL \subseteq QL$ of Lemma 1 is also satisfied:

$$
fL = (R\widehat{\Phi}L \cap \pi)L = R\widehat{\Phi}L \cap \pi L = R\widehat{\Phi}L = QL.
$$

qed

Even though it is a sufficient condition, the condition provided by Proposition 5 is still complex. The following proposition provides a simpler, if stronger, condition of recoverability preservation. Most importantly, the sufficient condition provided by this proposition satisfies our intuition that a past function preserves recoverability if it defines a finer domain partition than the specification.

Proposition 6 *Given a specification R and a future function Φ, if R is regular and the following conditions are satisfied*

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

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

Proof. The proof is fairly straightforward, and proceeds by successive implications, starting from the condition of regularity of R.

$$
\begin{array}{ll}
 & R\widehat{R}R \subseteq R \\
\Rightarrow & \qquad\qquad \{ \text{ monotonicity } \} \\
 & R\widehat{R}R\widehat{\Phi} \subseteq R\widehat{\Phi} \\
\Rightarrow & \qquad\qquad \{ \text{ hypothesis } \} \\
 & \pi\widehat{\pi}R\widehat{\Phi} \subseteq R\widehat{\Phi} \\
\Rightarrow & \qquad\qquad \{ \text{ lattice theory } \} \\
 & (R\widehat{\Phi}L \cap \pi)(R\widehat{\Phi}L \cap \pi)\,\hat{}\ R\widehat{\Phi} \subseteq R\widehat{\Phi}. \\
\Rightarrow & \qquad\qquad \{ \text{ Proposition 5 and hypothesis } \} \\
 & \pi \text{ preserves recoverability with respect to } \Phi \text{ and } R.
\end{array}
$$

qed

Example. The following example shows that the second and third conditions of proposition 6 are not sufficient to ensure recoverability preservation. Consider the following values of R, π, and Φ on space $S = \{1, 2, 3\}$.

$R := \{(1,1),(1,2),(2,2),(2,3),(3,1),(3,3)\}$
$\pi := \{(1,1),(2,1),(3,1)\}$
$\Phi := I = \{(1,1),(2,2),(3,3)\}.$

Then, π and Φ are functions and $\pi L = L$, so that $R\widehat{\Phi}L \subseteq \pi L$. Also, $R\widehat{R} = L$, so that $\pi\widehat{\pi} \subseteq R\widehat{R}$. However,

$$\overline{(R\widehat{\Phi}L \cap \pi)\,\widehat{}\;\overline{R\widehat{\Phi}}}L = \overline{\widehat{\pi}\overline{R}}L = \{(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)\} \not\supseteq L.$$

Hence the condition for the existence of a solution r to $\pi r \sqsupseteq R\widehat{\Phi}$ (alternately, to $\pi \,\square\, r \sqsupseteq \kappa(R,\Phi)$) does not hold. Clearly, π does not preserve recoverability, since it maps all inputs to 1, causing a loss of information (about the input, and the output) that no recovery function can recover from. □

Example. We consider again the two examples we discussed above to introduce recoverability preservation, and we apply Proposition 6 to them.

- *Sorting Routine.* We consider the *selection sort* algorithm for sorting an array a of size n, and we let i be the index we use to this effect, which ranges down from $n-1$ to 1. We let R be defined as

  $$R = \{(s,s') | prm(a(s), a(s')) \wedge std(a(s'))\},$$

  where $prm(a,a')$ means that a and a' are permutations of each other, and $std(a)$ means that a is sorted. An analysis of the selection sort algorithm yields the following definition of the future function:

  $$\Phi = \{(s,s') | a(s')[1..i] = ord(a(s)[1..i]) \wedge a(s')[i+1..n] = a(s')[i+1..n]\},$$

  where $ord(a)$ refers to the ordered permutation of a. The clauses of Proposition 6 can be written as follows:

  1. $R\widehat{R}R \subseteq R$.
  2. $R\widehat{\Phi}L \subseteq \pi L$.
  3. $\mu(\pi) \subseteq \mu(R)$.

  The first clause stems readily from the observation that R is deterministic. The second clause provides that π must be total, since the left term yields L. The third clause is interpreted as:

  $$\mu(\pi) \subseteq \{(s,s') | prm(a(s), a(s'))\}.$$

  So long as π distinguishes (i.e., maps to distinct outputs) arrays that are not permutations of each other, it preserves recoverability. Note that if instead of R we choose specification R' defined by

  $$R' = \{(s,s') | std(a(s'))\},$$

  then we find that R' is regular and that $\mu(R') = L$, hence any total function preserves recoverability. This, again, is intuitively acceptable, since specification R' makes no reference to the initial state, whence there is no initial state information to lose; any intermediate state is recoverable —all we have to do is sort the array (even if it has nothing to do with the original array).

- *Mod program.* We let the space be the set of natural numbers and R specify that we must compute 5 plus the remainder by 6 of the argument, i.e.,

  $$R = \{(s,s') | s' = 5 + (s \;\mathbf{mod}\; 6)\}.$$

  Further, we let the future function be defined as

  $$\Phi = \{(s,s') | s' = s + 5\}.$$

Considering again the clauses of Proposition 6, we find that the first clause stems readily from the property that R is deterministic, and that the second clause imposes that π must be total. As for the third clause, we find:

$$\mu(\pi) \subseteq \{(s, s')|(s \mathbf{\ mod\ } 6) = (s' \mathbf{\ mod\ } 6)\}.$$

The partition by π of its domain must be finer (or as fine as) the congruence relation modulo 6 over the set of natural numbers. Any past function that remembers the equivalence class (modulo 6) of the initial state preserves recoverability. Examples of functions that are recoverability-preserving include:

– $\pi = \{(s, s')|s' = 1 + s \mathbf{\ mod\ } 6\}$.
– $\pi = \{(s, s')|s' = s \mathbf{\ mod\ } 12\}$.
– $\pi = \{(s, s')|s' = (1 + s) \mathbf{\ mod\ } 6\}$.

Note that the determination of whether past function π does preserve recoverability or not depends on $\mu(\pi)$, and does not involve π other than through its nucleus. This means that the recoverability preservation of π depends on how π partitions its domain, but does not depend on what value π assigns to each partition of its domain, which is quite intuitive: any deviation in the way π maps images to arguments can be corrected by a recovery routine, hence does not affect recoverability preservation.

□

4 Conclusion

In [22] we have 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 sub-specifications 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, 26]. 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 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 [22], 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, 14]. 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. Faults that 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).

As for extensions of this work, we envisage the following theoretical extension:

- *Partial Recoverability Preservation*. This idea is best illustrated on an example, the modulo example. Imagine that the past function must compute $(s \mathbf{~mod~} 6)$ for a positive integer s given as input. We consider the following faulty past functions:

  - $\Pi = \{(s,s')|s' = s \mathbf{~mod~} 18\}$. This past function preserves recoverability. Recovery routine: $r = \{(s,s')|s' = s \mathbf{~mod~} 6\}$.
  - $\Pi = \{(s,s')|s' = (s+1) \mathbf{~mod~} 6\}$. This past function preserves recoverability. Recovery routine: $r = \{(s,s')|s' = (s+5) \mathbf{~mod~} 6\}$.
  - $\Pi = \{(s,s')|s' = s \mathbf{~mod~} 3\}$. In this case, even though the past function does not preserve recoverability, it still contains some partial information about the correct state: For example, if we find that $(s \mathbf{~mod~} 3) = 2$, we can infer that $(s \mathbf{~mod~} 6)$ is either 2 or 5. We say that this past function preserves partial recoverability.

  Our goal is to characterize the property of partial recoverability preservation, and eventually to quantify the extent of recoverability that has been preserved. Also, we wish to explore possible (probabilistic) recovery routines for such cases.

These issues are currently under investigation.

References

[1] A. Avizienis. The n-version approach to fault tolerant software. *IEEE Trans. on Software Engineering*, 11(12), December 1985.

[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. 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.

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

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

[17] 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.

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

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

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

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

[22] 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.

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

[24] C.C. Morgan. *Programming from Specifications*. International Series in Computer Sciences. Prentice Hall, London, UK, 1998.

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

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