

Ali Mili · Frederick Sheldon · Fatma Mili  
Jules Desharnais

## Recoverability preservation: a measure of last resort

Received: 28 October 2004 / Accepted: 13 January 2005 / Published online: 11 March 2005  
© Springer-Verlag 2005

**Abstract** Recoverability preservation is the property of a system to maintain recoverability even when it does not maintain correctness; recoverability, in turn, is the property of a system to avoid failure, even when system states have errors. In this paper, we argue that fault tolerance techniques could be more streamlined, less intrusive, and more effective if they focused on the criterion of recoverability preservation instead of the traditional criterion of correctness preservation. To this effect, we briefly introduce, motivate, illustrate, and analyze the concept of recoverability preservation, then we explore some of its applications.

**Keywords** Programming calculi · Relational mathematics · System fault tolerance · Recoverability preservation · Recovery routine

### 1 Motivation: recoverability preservation as a minimal requirement

It is common to distinguish between three broad families of methods for dealing with the presence and manifestation of faults in complex systems:

- *Fault avoidance*, which is based on the premise that it is possible to avoid faults in the design of complex systems and takes steps to design fault-free systems.
- *Fault removal*, which concedes that fault avoidance is unrealistic for complex systems and takes steps to identify and remove faults after system design.
- *Fault tolerance*, which concedes that both fault avoidance and fault removal are unrealistic and takes steps to prevent system faults from causing system failure.

In this paper, we focus on fault tolerance, which we analyze in light of the concept of recoverability preservation. For the purposes of this discussion, we adopt the fault tolerance terminology and definitions of Laprie [16]:

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**.

Fault tolerance refers to the set of measures and provisions that a system makes to avoid failure after faults have caused errors. Steps to provide fault tolerance include:

- *Error detection*, which consists in checking some correctness conditions during the execution of the system.
- *Damage assessment*, which consists in assessing the extent of the damage sustained by the system state, so as to determine an optimal / adequate recovery action.
- *Error recovery*, which consists in retrieving a correct state either from the current state (forward error recovery) or from a previously saved state (backward error recovery) and resuming the computation.

---

A. Mili (✉)  
College of Computing Science,  
New Jersey Institute of Technology,  
Newark, NJ 07102-1982, USA  
E-mail: mili@cis.njit.edu

F. Sheldon  
U.S. DOE Oak Ridge National Lab,  
P.O. Box 2008, MS 6085, 1 Bethel Valley Rd.,  
Oak Ridge, TN 37831-6085  
E-mail: sheldonft@ornl.gov

F. Mili  
School of Engineering and Computer Science,  
Oakland University,  
Rochester, MI 48309-4401,  
E-mail: mili@oakland.edu

J. Desharnais  
Département d'Informatique et GL,  
Université Laval,  
Québec PQ G1K 7P4 Canada  
E-mail: Jules.Desharnais@ift.ulaval.ca

- *Fault diagnosis and removal*, which (unlike all three of the other steps) is carried out offline and consists in identifying and removing the fault that has (presumably) caused the error.

Traditionally, fault tolerance techniques that focus on correctness present some inherent weaknesses, which we formulate by the following premises:

- *Trigger-happy error detection*. The focus on correctness leads us to trigger fault tolerance measures in cases when that is not necessary. We argue that, while the discovery of errors should always trigger fault removal, it does not have to systematically trigger error recovery.
- *Tedious error recovery*. The focus on correctness leads us to require that recovered states be correct, which is often unnecessary. We argue that error recovery does not have to produce an error-free state and that it is possible to avoid failure without retrieving a strictly correct state.
- *Inefficient checkpointing*. The focus on correctness mandates that we maintain copies of past correct states, at the cost of much time and space overhead. We argue that avoiding failure does not require such radical measures.
- *Panic-driven damage assessment*. The focus on correctness leads us to declare failure too early, as soon as we determine that we cannot retrieve / recover a correct state. We argue that a state can be severely damaged and still contain enough information to avoid failure.

The concept of *recoverability preservation*, which we discuss in this paper, can be introduced intuitively as part of a hierarchy of properties, ranked in decreasing order of correctness:

- *Correctness*, i.e., a system's ability to operate without producing errors.
- *Maskability*, i.e., a system's ability to avoid failure spontaneously (without recovery action) even when errors arise.
- *Recoverability preservation*, i.e., a system's ability to avoid failure, possibly by means of recovery action, even when errors arise.
- *Partial recoverability preservation*, i.e., a system's ability to reduce the likelihood of failure, by means of probabilistic recovery action, even when errors arise.

Using this hierarchy, we submit the following premises as guidelines on how to enhance the fault tolerance capability of a system in a lean, efficient, nonintrusive manner.

- We argue that the important test, for the purposes of error detection, is not whether a state is correct, but whether a state is maskable.
- We argue that the most important test, for the purposes of damage assessment, is whether the current state is recoverable, possibly whether it is partially recoverable.
- We argue that the purpose of a recovery routine is not to produce a correct state but only to produce a maskable state.
- We argue for the need to derive recovery routines (deterministic recovery, probabilistic recovery) by calculation rather than by inspection using the parameters of the system at hand.

- For extra generality, we argue that maskability and recoverability should be defined, not with respect to the expected function of the system, but rather with respect to the specification that the system is expected to satisfy. By and large, we can make the system fault tolerant, not with respect to the overall specification, but rather with respect to a selected subspecification that represents a property of interest (e.g., safety).
- We argue in favor of forward error recovery rather than backward error recovery; interestingly, this restriction does not in fact exclude backward error recovery but rather formulates error recovery in a way that encompasses both forms. Backward error recovery on a state  $s$  can be formulated as forward error recovery on a compound state  $\langle s, \bar{s} \rangle$ , where  $\bar{s}$  is the copy of  $s$  that is used to store checkpoint values of the state.
- We argue that fault tolerance should be used with correctness proofs, as some functional properties are better verified statically at design time (hence candidates for correctness verification) and others are better verified dynamically at run time (hence candidates for fault tolerance methods) [17, 18].
- We argue that fault tolerance methods should be modeled using the same mathematics as program verification and programming calculi to enable their deployment in concert, as advocated above.

We believe that the combination of these premises produces fault tolerance methods that are reasoned in their reaction (do not rush into panic mode [24]), measured in their response (do only as much as is needed, not much more), and sparing in their needs (obviate many of the drawbacks of checkpointing in terms of time and space overheads). Recoverability preservation plays a role in the formulation and rationalization of the proposed approach.

## 2 Recoverability preservation: intuitive illustrations

In this section, we introduce recoverability preservation by means of a simple/simplistic example. We wish to draw the reader's attention to the extent to which a function may deviate from its correct behavior but still preserve recoverability. As background for our discussions, we briefly introduce some mathematical notation. A relation  $R$  on set  $S$  is a subset of  $S \times S$ . Constant relations on  $S$  include the identity relation, denoted by  $I$ , and the total relation, denoted by  $L$ . Operations on relations include, in addition to the set theoretic operations of union and intersection, the *inverse*, defined by

$$\widehat{R} = \{(s, s') \mid (s', s) \in R\},$$

the *complement*, defined by

$$\overline{R} = L/R \text{ or } L - R$$

and the *composition*, defined by

$$R \circ R' = \{(s, s') \mid \exists t : (s, t) \in R \wedge (t, s') \in R'\}.$$

When no ambiguity arises, we may denote  $R \circ R'$  by  $RR'$ . A relation  $R$  is said to be total if and only if  $RL = L$  or, equivalently, if  $I \subseteq R\widehat{R}$ . A relation  $R$  is said to be *deterministic* if and only if  $\widehat{RR} \subseteq I$ . When a relation is deterministic, we say that it is a *function*. Given a total function  $F$  on  $S$ , we know that  $F\widehat{F}$  is an equivalence relation (i.e.,  $F\widehat{F} \subseteq \widehat{F\widehat{F}}$ ,  $I \subseteq F\widehat{F}$ , and  $F\widehat{F}(F\widehat{F}) \subseteq F\widehat{F}$ ). The equivalence classes of this relation are called the *level sets* of  $F$  [20, 19]. We say that a relation  $R$  is regular if and only if  $R\widehat{RR} = R$ .

Given a program or program part  $P$  on space  $S$ , we denote by  $[P]$  the function that  $P$  computes on its space, i.e., the mapping that it defines between its initial states and its final states. When this raises no ambiguity, we may, for the sake of convenience, confuse program  $P$  with its function  $[P]$ .

We consider the space  $S$  defined by an integer variable  $x$ , and we consider the following simple program

$P; L : F,$

where  $L$  is a label and  $P$  (past) and  $F$  (future) are defined by their (expected) functions as follows:

$$[P] = x \bmod 6,$$

$$[F] = x \bmod 9 + 12.$$

If the computation starts with initial state  $x_0$ , then at label  $L$  we must have state  $(x_0 \bmod 6)$ . This is the only correct state at label  $L$ .

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

$$[P_1] = x \bmod 6 + 18,$$

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  $([P] = x \bmod 6)$  it computes

$$[P_2] = x \bmod 12,$$

then the states that  $P_2$  computes are not even maskable, but they are recoverable, in the following sense: If we know what  $(x \bmod 12)$  is, 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  $([P] = x \bmod 6)$  it computes

$$[P_3] = x \bmod 3,$$

then the states that  $P_3$  computes are not even recoverable, but they are *partially recoverable*, in the following sense: If we know what  $(x \bmod 3)$  is, we may not know exactly what  $(x \bmod 6)$  is, 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* that 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  $([P] = x \bmod 6)$  it computes

$$[P_4] = x \bmod 7$$

then the states that  $P_4$  produces are neither recoverable nor partially recoverable, for the following reason: knowing  $(x \bmod 7)$  gives us no information whatsoever on the value of  $(x \bmod 6)$ .

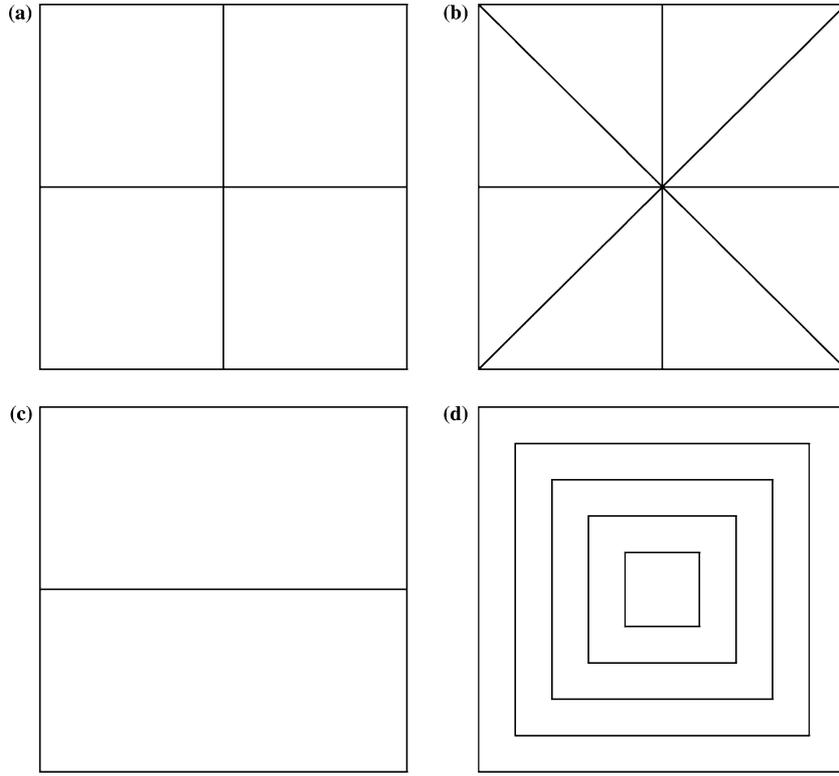
A superficial, intuitive look at this example seems to indicate that a function  $P'$  preserves recoverability with respect to an ideal function  $P$  if the level sets of  $P'$  refine (in the sense of: define a finer partition) the level sets of  $P$ ; we will revisit this characterization later, in Proposition 5. 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.

The condition that the level sets of  $P'$  refine the level sets of  $P$  has a simple intuitive interpretation: if  $P$  and  $P'$  are total, this condition is necessary and sufficient for the existence of a total function  $r$  that satisfies the equation

$$P' \circ r = P.$$

The proof of sufficiency can be articulated as follows:

$$\begin{aligned} & P'\widehat{P}' \subseteq P\widehat{P} \\ \Rightarrow & \quad \{\text{multiplying both sides by } P\} \\ & P'\widehat{P}'P \subseteq P\widehat{P}P \\ \Rightarrow & \quad \{\text{because } P \text{ is a function, } \widehat{P}P \subseteq I\} \\ & P'\widehat{P}'P \subseteq P \\ \Rightarrow & \quad \{\text{Let } r' \text{ be a function such that } r' \subseteq \widehat{P}'P \wedge \\ & \quad r'L = \widehat{P}'L\} \\ & \exists r': P'r' \subseteq P \\ \Rightarrow & \quad \{\text{let } r = r' \cup (I \cap \overline{r'L}), \text{ then } rL = L \text{ and} \\ & \quad Pr' = Pr\} \\ & \exists r: P'r \subseteq P \wedge rL = L \\ \Rightarrow & \quad \{P'r \text{ and } P \text{ are both total functions}\} \\ & \exists r: P'r = P \wedge rL = L. \end{aligned}$$



**Fig. 1** Degrees of recoverability. **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

The proof of necessity can be articulated as follows:

$$\begin{aligned}
& \exists r: P'r = P \wedge rL = L. \\
& \Rightarrow \quad \{\text{substitution}\} \\
& P'r\widehat{P'r} = P\widehat{P} \\
& \Rightarrow \{\text{relational identity}\} \\
& P'r\widehat{r\widehat{P}'} = P\widehat{P} \\
& \Rightarrow \quad \{\text{because } r \text{ is total, } I \subseteq r\widehat{r}\} \\
& P'\widehat{P}' \subseteq P\widehat{P}.
\end{aligned}$$

The condition

$$\exists \text{ total function } r: P'r = P$$

can be interpreted as saying that if, by mistake, we applied function  $P'$  rather than  $P$ , we may recover the result of applying function  $P$  by applying function  $r$  after  $P'$ . If no such  $r$  exists, then nothing can be done after application of  $P'$  to retrieve the result of function  $P$ . This is a useful illustration of recoverability preservation.

Figure 1 illustrates the hierarchy between the various properties that we have discussed:

- Figure 1a represents the partition of the domain of  $P$  by the equivalence relation  $P\widehat{P}$ . This is the same partition as that defined by  $P_1\widehat{P}_1$ , which preserves maskability.
- Figure 1b 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  $P_2\widehat{P}_2$ , then we know what partition we are in by  $P\widehat{P}$ .

- Figure 1c 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  $P_3\widehat{P}_3$ , then we can infer a limited set of possible partitions by  $P\widehat{P}$ .
- Figure 1d 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 as to what partition by  $P$  we are in; the two partitions are perfectly orthogonal.

### 3 Characterizing recoverability preservation

For the sake of readability, we will keep our discussion free of detailed mathematics, referring the interested reader to [8]. We will instead present without proof some propositions that characterize correctness preservation, maskability preservation, and recoverability preservation, and try to interpret the results in terms of the illustrative examples presented in Sect. 2. We use homogeneous binary relations to represent specifications and program functions, and we use relational algebra as a tool for relational manipulations and representations. Our main sources are [4, 6, 23], though we may depart

from them in terms of notation, in trivial and recognizable ways.

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  $R' \sqsubseteq R$ . In conjunction with the refinement ordering, we introduce a compositionlike operator, which we denote by  $R \circ R'$ , refer to as the *monotonic composition*, and define by

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

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  $R \circ R' \sqsupseteq Q \circ Q'$ . We introduce two related divisionlike operations on relations that will play a crucial role in our subsequent discussions. Because the monotonic composition is not commutative (nor is the simple composition), we need two divisionlike 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

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

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

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

The kernel is due to [9]; it is similar to operators introduced by [1–3, 5, 12–15, 22]. When  $\Phi$  is a function, we know (from [9]) that  $R\widehat{\Phi} = \kappa(R, \Phi)$ .

From an intuitive standpoint, the most useful interpretation of the kernel of relation  $R$  with relation  $R'$  is that it is the least refined solution in  $X$  to the equation

$$X \circ R \sqsupseteq R'.$$

Likewise, the cokernel of  $R$  with relation  $R'$  can be interpreted as the least refined solution in  $X$  to the equation

$$R \circ X \sqsupseteq R'.$$

We consider a system/program structured as the composition of two sequential components, say  $\Pi$  and  $\Phi$ , and we let  $K$  be the label that we reach after completing  $\Pi$ . We are interested in the state of the program at label  $K$  for some initial state  $s_0$ ; equivalently, we are interested in the degree of correctness (or faultiness?) of the past function. Because we are investigating whether the past function is correct, we will distinguish between the *actual* past function  $\Pi$  and the *ideal/expected* past function  $\pi$ . In the sequel, we give definitions that characterize levels of correctness of a state  $s$  at label  $K$ ; then, for each level, we provide propositions that characterize past functions that are guaranteed to produce states at that level of correctness.

**Definition 2.** State  $s$  at label  $K$  is said to be correct for initial state  $s_0$  with respect to (ideal) past function  $\pi$  if and only if

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

An ( $n$  actual) past function  $\Pi$  is said to *preserve correctness* if and only if it produces nothing but correct states at label  $K$ .

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

**Proposition 1.** An actual past function  $\Pi$  is correctness preserving if and only if  $\Pi \sqsupseteq \pi$ .

If function  $\pi$  is total, then  $\Pi \sqsupseteq \pi$  is equivalent to  $\Pi = \pi$ ; the only way to refine a total function is to be equal to it.

**Definition 3.** A state  $s$  is said to be *maskable* at label  $K$  with respect to specification  $R$  for initial state  $s_0$  and future function  $\Phi$  if and only if

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

An  $n$  actual past function  $\Pi$  is said to *preserve maskability* if and only if it produces nothing but maskable states at label  $K$ .

We have the following proposition, which characterizes in closed form past functions that preserve maskability.

**Proposition 2.** An  $n$  actual past function  $\Pi$  preserves maskability at label  $K$  with respect to  $R$  if and only if

$$\Pi \sqsupseteq \kappa(R, \Phi).$$

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.

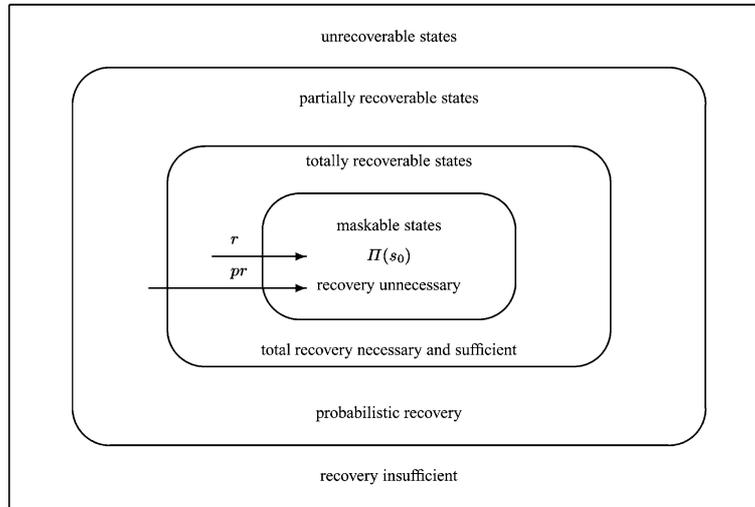
**Definition 4.** A state  $s$  is said to be *recoverable* with respect to  $R$  at label  $K$  for initial state  $s_0$  and future function  $\Phi$  if and only if there exists a function, say  $r$ , such that  $r(s)$  is maskable at label  $K$  for state  $s_0$  and function  $\Phi$  with respect to specification  $R$ . An  $n$  actual past function  $\Pi$  is said to *preserve recoverability* at label  $K$  if and only if it is guaranteed to produce nothing but recoverable states at label  $K$ .

Implicit in this definition is the requirement that  $r$  not be dependent on  $s$ , of course: the same function  $r$  must recover all states that are recoverable at the given label. We have the following proposition.

**Proposition 3.** Given specification  $R$  and future function  $\Phi$ , a past function  $\Pi$  preserves recoverability if and only if

$$HL \subseteq \Pi L \wedge L \subseteq \overline{(\widehat{HL} \cap \Pi)\widehat{HL}},$$

where  $H$  is short for  $\kappa(R, \Phi)$ .



**Fig. 2** A hierarchy of correctness levels

The specification of recovery routines is derived by computation (rather than by inspection) from the parameters of the program (the expected past function, the expected future function, the specification, etc.). The following proposition provides a specification of recovery routines.

**Proposition 4.** *If past function  $\Pi$  preserves recoverability with respect to future function  $\Phi$  and specification  $R$ , then*

$$\rho = \Gamma(\Pi, \kappa(R, \Phi))$$

*satisfies the equation:  $\Pi \circ \rho \sqsupseteq \kappa(R, \Phi)$ .*

In other words, if  $\Pi$  preserves recoverability and recovery routine  $r$  refines specification  $\rho$ , then (because  $\circ$  is monotonic with respect to refinement) the function  $\Pi \circ r$  preserves maskability (by virtue of Proposition 2.).

In the example discussed in Sect. 2, the reader may have gained the intuition that a function  $\Pi$  preserves recoverability with respect to an expected function  $\pi$  if and only if the equivalence relation  $\Pi \hat{\Pi}$  defines a finer partition of  $dom(\Pi)$  than the equivalence relation  $\pi \hat{\pi}$ . The following proposition, which provides a sufficient condition for recoverability preservation, reflects this intuition, though in more sophisticated/more general form.

**Proposition 5.** *Given a specification  $R$  and a future function  $\Phi$ , if  $R$  is regular and the following conditions are satisfied:*

$$R \hat{\Phi} L \subseteq \Pi L \text{ and } \Pi \hat{\Pi} \subseteq R \hat{R},$$

*then  $\Pi$  preserves recoverability with respect to future function  $\Phi$  and specification  $R$ .*

The reader may be interested to know that when  $R$  is regular,  $R \hat{R}$  is also an equivalence relation (as it is for deterministic relations); hence this proposition is still imposing a condition to the effect that the kernel of  $\Pi$  refines the kernel of the specification,  $R$ .

We have not yet explored characterizations of partial recoverability preservations, nor how to perform probabilistic

recovery; this is currently under investigation. Figure 2 shows the logical hierarchy between the various correctness levels of the past function; for the sake of completeness, we ought to also show the properties that represent maskability and recoverability with respect to a nondeterministic specification, but we forgo this to keep the figure simple.

## 4 Using recoverability preservation

In this section, we explore applications of recoverability preservation in the context of software fault tolerance.

### 4.1 Enhanced fault tolerance

The insights afforded by recoverability preservation allow us to sketch the outlines of a streamlined/economical method for fault tolerance, whose outline reads as follows:

```
if not maskable(s) then
  recoverymeasures(s);
```

where `recoverymeasures(s)` reads as follows

```
if recoverable(s)
  // proposition 5
  then
    deterministicrecovery(s)
    // proposition 4
  else
    if partiallyrecoverable(s)
      // section 2
    then
      probabilisticrecovery(s)
      // section 2
    else
      failure(s);
```

Function `recoverable(s)` would be derived from Proposition 5, using the specification  $R$  that represents the property we want to maintain; and function `deterministicrecovery(s)` would be derived from Proposition 4, for the same specification. Functions `partiallyrecoverable(s)` and `probabilisticrecovery(s)` are not specified as of yet; we depend on the discussions of Sect. 2 to convey our intention on these.

#### 4.2 Combining fault tolerance and fault avoidance

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 (especially for largely non-deterministic specifications), the former may be easier and produce more dependable conclusions. We are mindful of the complications that arise when we apply this kind of mathematics to large-scale, complex systems, and we are exploring methods to control the attending complexity by a variety of means (using refinement-compatible decomposition/composition operators, using induction, budgeting formality, etc.).

Also, we may break down a complex specification into simpler components and prove the program correct with respect to some components while making it fault tolerant with respect to others. In [18], we showed the complementarity of this heterogeneous approach.

#### 4.3 Cataloging fault modes

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 [7, 11]. 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) and faults that cannot be recovered from (and the flight control system is not to blame).

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,  $\Pi$ , is the function of the aggregate made up of the airframe and the sensors and actuators

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 Fig. 3.

- The future function,  $\Phi$ , is the function of the flight control software (FCS), which analyzes the state of the aircraft (represented by sensor outputs) and the pilot commands, as well as any navigation signals (e.g., ILS) and computes the actuator inputs (which are then fed to the actuators).
- The specification  $R$  represents a relation we wish to impose between the current state of the aircraft, navigation signals, and the pilot commands on the 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 times to ensure the safety of the flight.

The condition of recoverability preservation can be interpreted as the minimal requirement that the past function  $\Pi$  (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 an error is suspected. 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  $\pi$ . But under fault-prone conditions, this aggregate may produce a different function, say  $\Pi$ . In [10] we discuss how we can derive the specification of a behavioral envelope that captures all the possible functions defined by  $\Pi$  under a variety of precataloged fault modes. What Proposition 3. provides is the minimal requirement that  $\Pi$  must satisfy (refine) to ensure recoverability; as long as  $\Pi$  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  $\Pi$  does not refine the minimal requirements should not be supported, because it cannot be recovered from.

#### 4.4 Fail safety

Following the suggestion of Parnas [21], we have considered the integration of this hierarchy of recoverability levels with the concept of fail safety and have subscribed to the following premises:

- Safety is not fundamentally distinct from correctness, in the sense that it can be viewed as correctness of the system with respect to a specification that reflects a (minimal) safety property. While we denote the specification of the system as  $R$ , we refer to the specification of the safety property as  $T$ .

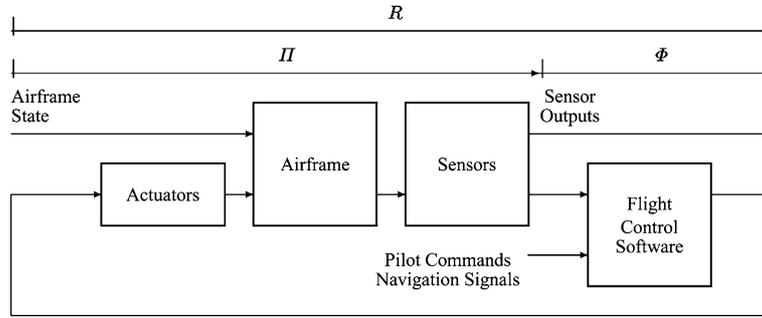


Fig. 3 Outline of a flight control loop

Correctness, $R$	$R$ -Maskability	$R$ -Recoverability	$R$ -Partial Recoverability	$R$ -Failure
$T$ , Safety				
$T$ -Maskability	$I$ Correctness	$\rho_R$ Correctness	$\beta_R$ Safety, Possible Correctness	$I$ Safety
$T$ -Recoverability		$\rho_R$ Correctness	$\rho_T \sqcup \beta_R$ Safety, Possible Correctness	$\rho_T$ Safety
$T$ -Partial Recoverability			$\beta_R \sqcup \beta_T$ Possible Safety, Possible Correctness	$\beta_T$ Possible Safety
$T$ -Failure				$I$ Failure

Fig. 4 Integrating correctness and safety concerns

- We assume that  $R$  refines  $T$ , i.e., that a correct system does not violate safety constraints.
- We assume, without proof, that the properties of maskability and recoverability are monotonic with respect to refinement, i.e., if they hold for a specification  $R$  they hold for any specification  $T$  that is refined by  $R$ . While this appears quite trivial, we have not proven it yet, though we expect it to hold.

A possible (simplistic) approach to this integration consists of revisiting procedure `recoverymeasures(s)` presented in Sect. 4.1 and replacing the clause `failure(s)` in this procedure by an altered version of `recoverymeasures(s)` in which all the conditions (maskability, recoverability, partial recoverability, etc.) are defined in terms of  $T$  rather than  $R$ . But this approach is not sufficiently integrated; we wish to juggle the concerns of correctness ( $R$ ) and safety ( $T$ ) simultaneously. To see how this can be done, we draw a plan of action, presented in Fig. 4, which considers all possible degrees of recoverability with respect to  $R$  and  $T$ . In each entry, we represent (by its specification) the suggested recovery action and (in the lower right corner of the entry) the outcome we expect from resuming execution after recovery. We will denote the specification of the recovery routine with respect to specification  $R$  by  $\rho_R$ ; its formula is given in Proposition 4.. Also, we will denote by  $\beta_R$  the specification of the probabilistic recovery with respect to specification  $R$ ; though we have no explicit formula for it, we assume that it is defined. For the sake of uniformity, we represent the case where no recov-

ery is necessary by the identity relation (specification). We assume that all the joins written in the table are defined (i.e., the specifications can be refined simultaneously); this matter is under investigation.

### 5 Prospects

As for the prospects for this work, we envision the following extensions:

- Characterizing the property of partial recoverability preservation by equivalent conditions and by simple sufficient conditions.
- Characterizing the specification of probabilistic recovery routines as a function of the degree of recoverability preservation.
- Exploring applications of these mathematics to assess their usefulness in practice.
- Combining and analyzing ideas of recoverability preservation with ideas of fail safety to derive a broad plan of action.

### References

1. Backhouse R, DeBruin P, Malcolm G, Voermans E, Van der Woude J (1990) A relational theory of data types. In: Proceedings of the workshop on constructive algorithms: the role of relations in program development, Hollum Ameland, Holland

2. Berghammer R, Schmidt G, Zierer H (1986) Symmetric quotients. Technical Report TUM-I8620, Technische Universität München, Munich
3. Berghammer R, Schmidt G, Zierer H (1989) Symmetric quotients and domain constructions. *Inf Process Lett* 33:163–168
4. Berghammer R, Schmidt G (1993) Relational specifications. In: Rauszer C (ed) *Proceedings of the XXXVIII Banach Center semester on algebraic methods in logic and their computer science applications*. Banach, vol 28, Warsaw, Poland, pp 167–190
5. Birkhoff G (1967) *Lattice theory*. American Mathematical Society, Providence
6. Brink Ch, Kahl W, Schmidt G (1997) *Relational methods in computer science*. Springer, Berlin Heidelberg New York
7. Cortellessa V, Mili A, Cukic B, Del Gobbo D, Napolitano M, Shereshevsky M (2000) Certifying adaptive flight control software. In: *Proceedings of ISACC 2000: the software risk management conference*, Reston, VA
8. DelGobbo D, Shereshevsky M, Cortellessa V, Desharnais J, Mili A (2005) A relational characterization of system fault tolerance. *Science of computer programming* (in press)
9. Desharnais J, Jaoua A, Mili F, Boudriga N, Mili A (1993) A relational division operator: the conjugate kernel. *Theor Comput Sci* 114:247–272
10. Del Gobbo D, Cukic B (2001) Validating on-line neural networks. Technical report, Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV
11. Del Gobbo D, Mili A (2001) Re-engineering fault tolerant requirements: a case study in specifying fault tolerant flight control systems. In: *Proceedings of the 5th IEEE international symposium on requirements engineering*, Toronto, pp 236–247
12. Hoare CAR, Hayes IJ, He JF, Morgan C, Roscoe AW, Sanders JW, Sorenson IH, Spived JM, Sufriñ B (1987) Laws of programming. *Commun ACM* 30(8):672–686
13. Hoare CAR, He JF (1986) The weakest prespecification. *Fundamentae Informaticae IX: Part I*: pp 51–58. *Part II*: pp 217–252
14. Jónsson B (1982) Varieties of relational algebras. *Algebra Universalis* 15:273–298
15. Josephs MB (1987) An introduction to the theory of specification and refinement. Technical Report RC 12993, IBM Corporation, Yorktown Heights, NY, USA
16. Laprie JC (1995) Dependability – its attributes, impairments and means. In: *Predictably dependable computing systems*. Springer, Berlin Heidelberg New York, pp 1–19
17. Lowry M, Boyd M, Kulkarni D (1998) Towards a theory for integration of mathematical verification and empirical testing. In: *Proceedings of the 13th IEEE international conference on automated software engineering*, Honolulu IEEE Computer Society, pp 322–331
18. Mili A, Cukic B, Xia T, Ben Ayed R (1999) Combining fault avoidance, fault removal and fault tolerance: an integrated model. In: *Proceedings of the 14th IEEE international conference on automated software engineering*, Cocoa Beach IEEE Computer Society, pp 137–146
19. Mills HD, Dyer M, Linger R (1987) Cleanroom software engineering. *IEEE Softw* 4(5):19–25
20. Mills HD, Linger RC, Hevner AR (1985) *Principles of information systems analysis and design*. Academic, New York
21. Parnas D (2004) Private correspondence. Technical report, University of Limerick, Ireland
22. Schmidt G, Stroehlein T (1990) *Relationen und Graphen*. Springer, Berlin Heidelberg New York
23. Schmidt G, Stroehlein T (1993) *Relations and graphs, discrete mathematics for computer scientists*. EATCS Monographs on theoretical computer science. Springer, Berlin Heidelberg New York
24. Selding PB (1996) Faulty software caused ariane 5 failure. *Space News* 7(25)