

Recoverability Preservation: A Measure of Last Resort

Ali Mili, Frederick Sheldon[†], Fatma Mili, Jules Desharnais

¹ College of Computing Science
New Jersey Inst. of Technology
Newark NJ 07102-1982, USA
mili@cis.njit.edu

² U.S. DOE Oak Ridge National Lab
PO Box 2008, MS 6085, 1 Bethel Valley Rd
Oak Ridge TN 37831-6085
sheldonft@ornl.gov

³ School of Engineering and Computer Science
Oakland University
Rochester MI 48309-4401
mili@oakland.edu

⁴ Département d'Informatique et GL
Université Laval
Québec PQ G1K 7P4 Canada
Jules.Desharnais@ift.ulaval.ca

Abstract. Traditionally, it is common to distinguish between three broad families of methods for dealing with the presence and manifestation of faults in digital (hardware or software) systems: Fault Avoidance, Fault Removal and Fault Tolerance. We focus on fault tolerance and submit that current techniques of fault tolerance would benefit from a better understanding of recoverability preservation, i.e. a system's ability to preserve recoverability even when/ if it does not preserve correctness. In this extended abstract, we briefly introduce the concept of recoverability preservation, discuss some preliminary characterizations of it, then explore possible applications thereof.

Keywords

Programming Calculi, Relational Mathematics, System Fault Tolerance, Fault, Error, Failure, Recoverability Preservation, Recovery Routine.

1 Introducing Recoverability Preservation

For the purposes of this extended abstract, we will introduce recoverability preservation by means of a simple/ simplistic example. We wish to draw the reader's attention on the extent to which a function may deviate from its correct behaviour but still preserve recoverability. 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

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

[†]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.

then the states that P_1 produces are not correct, but they are still maskable, since 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 is ($x \bmod 12$), we can derive ($x \bmod 6$). We say that P_2 *preserves recoverability* with respect to the expected past function P . It is possible to recover from errors caused by P_2 by simply applying ($\bmod 6$) to the current (potentially erroneous) state.

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

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

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

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

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. Notice that in this sample illustrative case, we have, for the sake of simplicity, neglected to factor in an important source of redundancy: the non-determinacy of the specification that the program at hand must satisfy. In effect we have assumed that the specification that the program must satisfy is the expected function of the program; had we chosen a less deterministic specification, say R (such that $[P; F]$ refines R), we would have highlighted further possibilities for deviating from the expected past function without jeopardizing recoverability preservation. This highlights our premise that recoverability preservation is a weak property, in the sense that it is possible for a program to deviate significantly from its intended function while still preserving recoverability, hence without jeopardizing the possibility of recovery.

2 Motivating Recoverability Preservation

For the purposes of this summary discussion, we adopt the 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 along 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.

- *Fault Diagnosis and Removal*, which (unlike all three other steps) is carried out off-line, and consists in identifying and removing the fault that has (presumably) caused the error.

We submit the following premises as guidelines of how to provide software applications with fault tolerance capabilities, in light of the properties of recoverability preservation.

1. 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, i.e. ultimately avoids failure.
2. We argue that the most important test, for the purpose of damage assessment, is whether the current state is recoverable, possibly whether it is partially recoverable.
3. We argue that the purpose of a recovery routine is not to produce a correct state but only to produce a maskable state.
4. 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.
5. 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 sub-specification that represents a property of interest (e.g. safety).
6. 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.
7. 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 are fault tolerance methods) [17, 18].
8. 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 [21]), 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 fault tolerance methods in terms of time and space overheads). Recoverability preservation plays a central role in the formulation and rationalization of our proposed approach.

3 Characterizing Recoverability Preservation

In this extended abstract, we cannot dwell on the detailed mathematics of recoverability preservation; the interested reader is referred to [8]. We will, however, present without proof some propositions that characterize correctness, maskability and recoverability preservation, and try to interpret the results in terms of the illustrative examples presented in section 1.

We use homogeneous binary relations to represent specifications and program functions, and we use relational algebra as a tool for relational manipulations and proofs. Our main sources are [4, 6, 20], 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

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

where L is the total relation. We abbreviate this property by $R \sqsupseteq R'$ or $R' \sqsubseteq R$. In conjunction with the refinement ordering, we introduce a composition-like operator, which we denote by $R \circ R'$, refer to as the *monotonic product*, 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

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

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

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

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

The kernel is due to [9]; it is similar to operators introduced by [1–3, 5, 12–15, 19].

We consider a system/ program structured as the composition of two sequential components, say P and F , and we let L be the label that we reach after completing P . We are interested in the state of the program at label L for some initial state s_0 ; equivalently, we are interested in the degree of correctness (or faultness?) of the past function.

Definition 2. State s at label L 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 label L , we say that we are observing an *error* at label L . Error detection relies on the condition of correctness to detect errors.

Definition 3. A state s is said to be maskable at label L for initial state s_0 and future function Φ 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 1. A state s is maskable at label L for initial state s_0 with respect to R if and only if

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

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 at label L 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 label L 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 label. We have the following proposition.

Proposition 2. Given specification R and future function Φ , a past function π preserves recoverability if and only if

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

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

The following proposition provides a specification of recovery routines. 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).

Proposition 3. 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: $\pi \square r \sqsupseteq \kappa(R, \Phi)$.

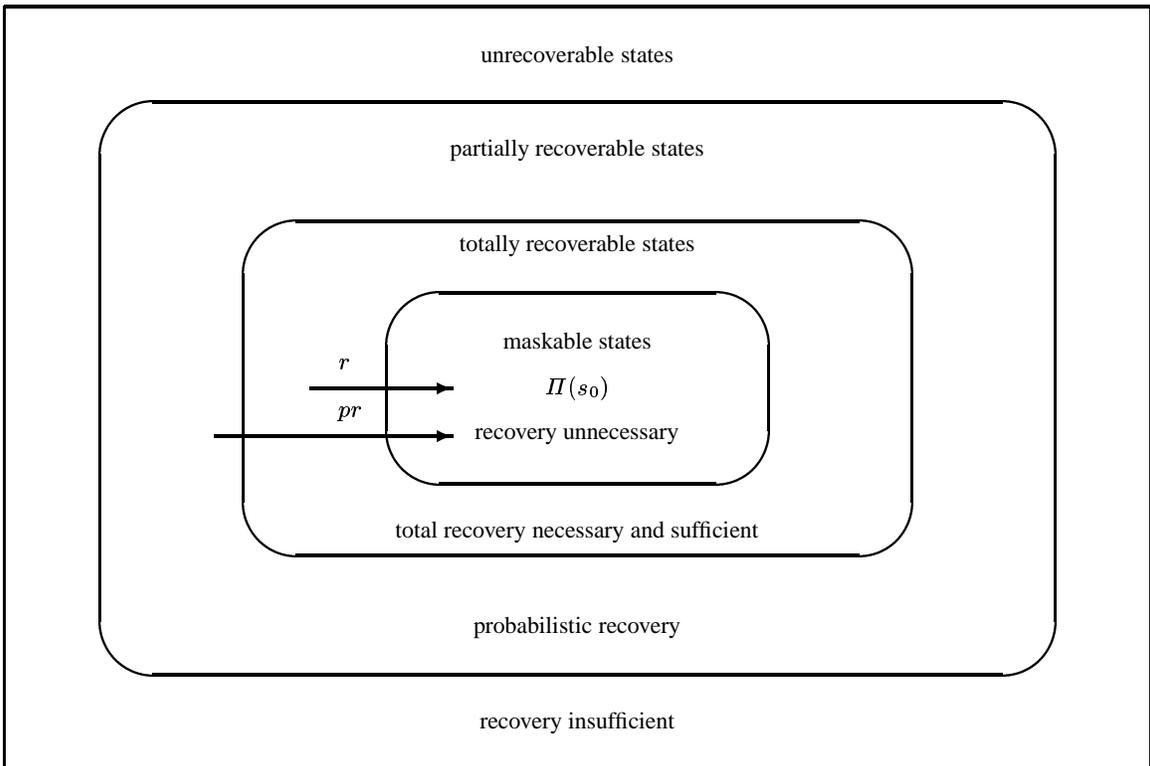


Fig. 1. A Hierarchy of Correctness Levels.

In the example discussed in section 1, the reader may have gained the intuition that a function π preserves recoverability with respect to an expected function \hat{H} if and only if the equivalence relation $\pi\hat{\pi}$ defines a finer partition of $dom(\pi)$ than the equivalence relation $\hat{H}\hat{H}$. The following proposition, which provides a sufficient condition for recoverability preservation, reflects this intuition, though in more sophisticated form.

Proposition 4. *Given a specification R and a future function Φ , 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 π preserves recoverability with respect to future function Φ and specification R .

We have not yet explored characterizations of partial recoverability preservations, nor how to perform probabilistic recovery; this is currently under investigation. Figure 1 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 non-deterministic 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. In this extended abstract, we keep our discussions at a very superficial level, barely getting into detail.

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 4
  then
    deterministicrecovery(s) // proposition 3
  else
    if partiallyrecoverable(s) // section 1
      then
        probabilisticrecovery(s) // section 1
      else
        failure(s);
```

Function `recoverable(s)` would be derived from Proposition 4, using the specification R that represents the safety property we want to maintain; and function `deterministicrecovery(s)` would be derived from Proposition 3, for the same specification. Functions `partiallyrecoverable(s)` and `probabilisticrecovery(s)` are not specified as of yet; we depend on the discussions of section 1 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, 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 had shown 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) from 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, 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 2.
- 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, as well as any navigation signals (e.g. ILS) 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, navigation signals, 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.

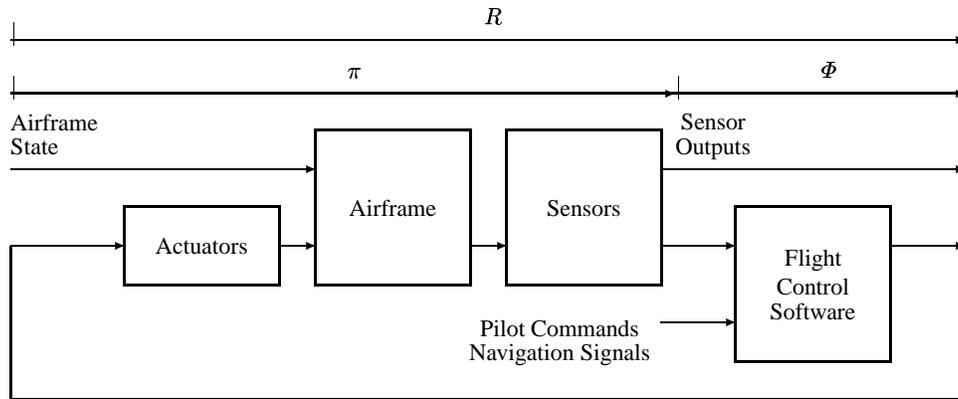


Fig. 2. Outline of a Flight Control Loop.

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 3, 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 3, 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 [10] 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 2 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.

5 Prospects

As for prospects of this work, we envisage 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.

References

1. 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.
2. R. Berghammer, G. Schmidt, and H. Zierer. Symmetric quotients. Technical Report TUM-I8620, Technische Universitaet Muenchen, Muenchen, Germany, 1986.
3. R. Berghammer, G. Schmidt, and H. Zierer. Symmetric quotients and domain constructions. *Information Processing Letters*, 33:163–168, 1989.
4. 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.
5. G. Birkhoff. *Lattice Theory*. American Mathematical Society, Providence, RI, 1967.
6. Ch. Brink, W. Kahl, and G. Schmidt. *Relational Methods in Computer Science*. Springer Verlag, January 1997.
7. 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.
8. Diego DelGobbo, Mark Shereshevsky, Vittorio Cortellessa, Jules Desharnais, and Ali Mili. A relational characterization of system fault tolerance. *Science of Computer Programming*, 2004.
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. 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.
11. 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.
12. C.A.R. Hoare and et al. Laws of programming. *Communications of the ACM*, 30(8):672–686, 1987.
13. 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.
14. B. Jónsson. Varieties of relational algebras. *Algebra Universalis*, 15:273–298, 1982.
15. M.B. Josephs. An introduction to the theory of specification and refinement. Technical Report RC 12993, IBM Corporation, July 1987.
16. J.C. Laprie. Dependability —its attributes, impairments and means. In *Predictably Dependable Computing Systems*, pages 1–19. Springer Verlag, 1995.
17. M. Lowry, M. Boyd, and D. Kulkarni. Towards a theory for integration of mathematical verification and empirical testing. In *Proceedings, 13th IEEE International Conference on Automated Software Engineering*, pages 322–331, Honolulu, HI, October 1998. IEEE Computer Society.
18. 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.

19. G. Schmidt and T. Stroehlein. *Relationen und Graphen*. Springer-Verlag, Berlin, Germany, 1990.
20. G. Schmidt and T. Stroehlein. *Relations and Graphs, Discrete Mathematics for Computer Scientists*. EATCS-Monographs on Theoretical Computer Science. Springer Verlag, 1993.
21. P. B. Selding. Faulty software caused ariane 5 failure. *Space News*, 7(25), June 1996.