A Survey of Linguistic Structures for Application-Level Fault Tolerance

VINCENZO DE FLORIO and CHRIS BLONDIA

University of Antwerp and Interdisciplinary Institute for BroadBand Technology

6

Structures for the expression of fault-tolerance provisions in application software comprise the central topic of this article. Structuring techniques answer questions as to how to incorporate fault tolerance in the application layer of a computer program and how to manage the fault-tolerant code. As such, they provide the means to control complexity, the latter being a relevant factor for the introduction of design faults. This fact and the ever-increasing complexity of today's distributed software justify the need for simple, coherent, and effective structures for the expression of fault-tolerance in the application software. In this text we first define a "base" of structural attributes with which application-level fault-tolerance structures can be qualitatively assessed and compared with each other and with respect to the aforementioned needs. This result is then used to provide an elaborated survey of the state-of-the-art of application-level fault-tolerance structures.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures—*Languages*; *domain-specific architectures*; D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Fault tolerance*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software development*; *software maintenance*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Portability*

General Terms: Design, Languages, Reliability

Additional Key Words and Phrases: Language support for software-implemented fault tolerance, separation of design concerns, software fault tolerance, reconfiguration and error recovery

ACM Reference Format:

De Florio, V. and Blondia, C. 2008. A survey of linguistic structures for application-level fault tolerance. ACM Comput. Surv. 40, 2, Article 6 (April 2008), 37 pages DOI = 10.1145/1348246.1348249 http://doi.acm.org/10.1145/1348246.1348249

1. INTRODUCTION

Research in fault tolerance has focused for decades on *hardware* fault tolerance, that is, on devising a number of effective and ingenious hardware solutions to cope with physical faults. For several years this approach was considered as the only effective

Authors' addresses: V. De Florio (corresponding author) and C. Blondia, Department of Mathematics and Computer Science, Performance Analysis of Telecommunications Systems Group, University of Antwerp, Middelheimlaan 1, 2020 Antwerp, Belgium, Interdisciplinary institute for BroadBand Technology, Gaston Crommenlaan 8, 9050 Ghent-Ledeberg, Belgium; email: Vincenzo.deflorio@ua.ac.be; chris.blondia@ua.ac.be.

solution to reach the required availability and data integrity demands of ever-more complex computer services. We now know that this is not true. Hardware fault tolerance is an important requirement to tackle the problem, but cannot be considered as the only option: Adequate tolerance of faults and end-to-end fulfilment of the dependability design goals of a complex software system must include means to avoid, remove, or tolerate faults that operate at all levels, *including the application layer*.

While effective solutions have been found, such as, for the hardware-[Pradhan 1996], operating system-[Denning 1976], or middleware [OMG 1998] layers, the problem of an effective system structure to express fault-tolerance provisions in the application layer of computer programs is still open. To the best of our knowledge, no detailed critical survey of the available solutions exists. Through this article the authors attempt to fill in this gap. Our target topic is linguistic structures for application-level fault tolerance, so we do not tackle herein other important approaches that do not operate at application level, such as the fault-tolerance models based on transparent task replication [Guerraoui and Schiper 1997], (e.g., CORBA-FT [OMG 1998]). Likewise, this text does not include approaches such as that in Ebnenasir and Kulkarni [2004], where the focus is on automating the transformation of a given fault-intolerant program into a fault-tolerant one. The reason behind this choice is that, due to their exponential complexity, those methods are currently only effective when the state space of the target program is very limited [Kulkarni and Arora 2000].

Another important goal of this text is to pinpoint the consequences of inefficient solutions to the aforementioned problem, as well as to increase awareness of a need for its optimal solution: Indeed, the current lack of a simple and coherent system structure for software fault-tolerance engineering (providing the designer with effective support towards fulfilling goals such as maintainability, reusability, and service portability), manifests itself as a true bottleneck for system development.

The structure of this work is as follows: In Section 2 we explain the reasons behind the need for system-level fault tolerance. Therein we also provide a set of key desirable attributes for a hypothetically perfect linguistic structure for application-level fault tolerance (ALFT). Section 3 is a detailed survey of modern available solutions, each qualitatively assessed with respect to our set of attributes. Some personal considerations and conjectures on what is missing and possible ways to achieve it are also part of that section. Section 4 finally summarizes our conclusions and provides a comparison of the reviewed approaches.

2. RATIONALE

In the early days of modern computing it was to some extent acceptable that outages and wrong results occurred rather often[1] (since the main role of computers was basically that of a fast solver of numerical problems). Today the criticality associated with many, however, tasks dependent on computers requires strong guarantees for properties such as availability and data integrity.

A consequence of this growth in complexity and criticality is the need for:

—techniques to assess and enhance, in a justifiable way, the reliance placed on services provided by computer systems; and

[1]This excerpt from a report on the ENIAC activity [Weik 1961] gives an idea of how dependable computers were in 1947: "[P]ower line fluctuations and power failures made continuous operation directly off transformer mains an impossibility . . . down times were long; error-free running periods were short." After many considerable improvements, still "trouble-free operating time remained at about 100 hours a week during the last 6 years of the ENIAC's use," namely, an availability of about 60%!

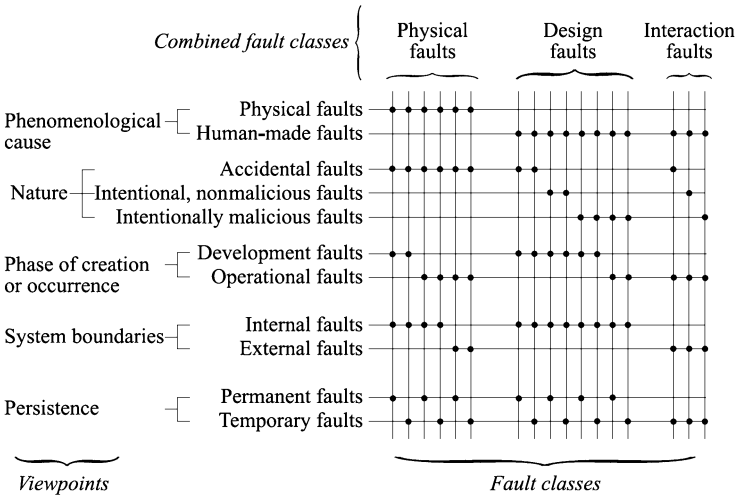

Fig. 1. Laprie's fault classification scheme.

—techniques to lessen the risks associated with computer failures, or at least to bound the extent of their consequences.

This article focuses in particular on *fault tolerance*, that is, how to ensure a service capable of fulfilling the system's function, even in the presence of "faults" [Avižienis et al. 2004a, 2004b].

A fault is a defect, imperfection, or lack in a system's hardware or software component. It is generically defined as the adjudged or hypothesized cause of an error. Faults can have their origin within the system boundaries (internal faults) or from without, namely, in the environment (external faults). In particular, an internal fault is said to be active when it produces an error, and dormant (or latent) when it does not. A dormant fault becomes an active fault when activated by either its process or the environment. Fault latency is defined as either the length of time between the occurrence of a fault and the appearance of the corresponding error, or the length of time between the occurrence of a fault and its removal.

Faults can be classified according to the so-called viewpoints [Laprie 1998, 1995, 1992] of phenomenological cause, nature, phase of creation or occurrence, situation with respect to system boundaries, and persistence. Not all combinations can give rise to a fault class; Laprie's process only defines 17 fault classes, summarized in Figure 1. These classes have been further partitioned into three groups collectively known as combined fault classes. We next describe them.

Physical Faults. These fault classes are comprised of the following.

(1) *Permanent, Internal, Physical Faults*. This class concerns those faults that have their origin within hardware components and are continuously active. A typical example is given by the fault corresponding to a worn-out component.

(2) *Temporary, Internal, Physical Faults (also known as Intermittent Faults)* [Bondavalli et al. 1997]. These are typically internal, physical defects that become active depending on a particular pointwise condition.

(3) *Permanent, External, Physical Faults*. These are faults induced on the system by the physical environment.

(4) *Temporary, External, Physical Faults (also known as Transient Faults)* [Bondavalli et al. 1997]. These are faults induced by environmental phenomena, such as EMI.

Design Faults. The next category of fault class consits of those faults introduced at design time.

(1) *Intentional, though Not Malicious, Permanent / Temporary Design Faults*. These are basically tradeoffs introduced at application-layer design time. A typical example is insufficient dimensioning (underestimations of the size of a given field in a communication protocol[2].)

(2) *Accidental, Permanent, Design Faults (also called Systematic Faults, or Bohrbugs)*. These are flawed algorithms that systematically make the same errors in the presence of the same input conditions and initial states; for instance, an unchecked divisor can result in a division-by-zero error.

(3) *Accidental, Temporary Design Faults (known as Heisenbugs, as in "bugs of Heisenberg", for their elusive character).* While systematic faults have an evident, deterministic behavior, these bugs depend on subtle combinations of the system state and environment.

Interaction Faults. The final fault class is comprised of those faults that occur during interactions.

(1) *Temporary, External, Operational, Human-Made Accidental Faults.* These include operator faults in which an operator does not correctly perform his or her role in system operation.

(2) *Temporary, External, Operational, Human-Made Nonmalicious Faults.* These faults arise through "neglect, interaction, or incorrect use problems" [Sibley 1998]. Examples include poorly chosen passwords and bad system parameter setting.

(3) *Temporary, External, Operational, Human-Made Malicious Faults.* This class includes the so-called malicious replication faults, namely faults that occur when the replicated information in a system becomes inconsistent, for example, because processes that are supposed to provide identical results no longer do so.

2.1. A Need for Software Fault-Tolerance

Research in fault tolerance concentrated for many years on hardware fault tolerance, that is, on devising a number of effective and ingenious hardware structures to cope with faults [Johnson 1989]. For some time this approach was considered as the only one needed in order to reach the availability and data integrity demands of modern, complex computer services. Probably the first researcher to realize this was far from assumption true was B. Randell, who in Randell [1975] questioned hardware fault tolerance as the only approach to pursue; in the cited paper he states:

[2]A noteworthy example is given by the bad dimensioning of IP addresses which gave raise to IPv6.

> Hardware component failures are only *one* source of unreliability in computing systems, decreasing in significance as component reliability improves, while software faults have become increasingly prevalent with the steadily increasing size and complexity of software systems.

Indeed, most of the complexity supplied by modern computing services lies in their software, rather than hardware, layer [Lyu 1998a, 1998b; Huang and Kintala 1995; Wiener 1993; Randell 1975]. This phenomenon could only be reached by exploiting a powerful conceptual tool for managing complexity in a flexible and effective way, that is, by devising hierarchies of sophisticated abstract machines [Tanenbaum 1990]. This translates into implementing software with high-level computer languages lying on top of other software strata: the device-driver layers, basic services kernel, operating system, runtime support of the involved programming languages, and so forth.

Partitioning the complexity into stacks of software layers allowed the implementors to focus exclusively on the high-level aspects of their problems, and hence allowed them to manage greater and greater degrees of complexity. However, though made transparent, this complexity is still part of the overall system being developed. A number of complex algorithms are executed by the hardware at the same time, resulting in the simultaneous progress of many system states under the hypothesis that neither the involved abstract machine nor the actual hardware will be affected by faults. Unfortunately, since in real life faults do occur, the corresponding deviations are likely to jeopardize the system's function. Moreover, faults can propagate from one layer to another, unless appropriate means are taken to either remove or tolerate these faults, or to avoid creating them in the first place. Furthermore, faults may also occur in the *application layer*, that is, in the abstract machine on top of the software hierarchy.[3] These faults, possibly having their origin at design time, during operation, or while interacting with the environment, do not differ in terms of their consequences from those faults originating, in the hardware or operating system. A well-known example is the case of Ariane 5 flight 501 [Inquiry Board Report 1996], in which the consequences of a fault in the application ultimately caused the system to crash. In general we observe that the higher the level of abstraction, the higher the complexity of the algorithms that come into play and consequent error proneness of the involved (real or abstract) machines. As such, we conclude that full tolerance of faults and complete fulfilment of the dependability design goals of a complex software application must include means to avoid, remove, or tolerate faults working at all, including application, layers. This article focuses on runtime detection and recovery of faults through mechanisms either residing in or cooperating with the application layer.

2.2. Software Fault-Tolerance in the Application Layer

The need for software fault tolerance provisions, located in the application layer, is supported by studies showing that the majority of failures experienced by modern computer systems are due to software faults, including those located in the application layer [Lyu 1998a, 1998b; Avižienis et al. 2004a, 2004b]; for instance, NRC reported that 81% of the total number of outages of US switching systems in 1992 were due to software faults [NRC 1993]. Moreover, modern application software systems are increasingly networked and distributed. Such systems, such as client-server applications, are often characterized by a loosely coupled architecture whose global structure is in general

[3]In what follows, the application layer is intended as the programming and execution context in which a complete, self-contained program that performs a specific function directly for the user is expressed or running.

more prone to failures.[4] Due to the complex and temporal nature of the interleaving of messages and computations in distributed software systems, no amount of verification, validation, and testing can eliminate all faults in an application and give complete confidence in the availability and data consistency of applications of this kind [Huang and Kintala 1995]. Under these assumptions, the only alternative (and effective) means for increasing software reliability is that of incorporating in the application software provisions for software fault tolerance [Randell 1975].

Another argument that justifies the addition of software fault tolerance means in the application layer is given by the widespread adoption of reusable software components. Approaches such as object orientation, component-based middleware, and service orientation have provided the designer with effective tools to compose systems out of, for example, COTS object libraries, third-party components, and Web services. For instance, many object-oriented applications are indeed built from reusable components, the sources of which are unknown to the application developers. The aforementioned approaches fostered the capability of dealing with higher levels of complexity in software and at the same time eased and therefore encouraged software reuse. This has a big, positive impact on development costs, but turns the application into a sort of collection of reused, preexisting "blocks" made by third parties. The reliability of these components and therefore their impact on the overall reliability of the user application is often unknown, to the extent that Green defines as "art" creating reliable applications using off-the-shelf software components [Green 1997]. The case of the Ariane 501 flight is a well-known example that shows how improper reuse of software may have severe consequences [Inquiry Board Report 1996].[5]

However, probably the most convincing argument for not excluding the application layer from a fault tolerance strategy is the so-called "end-to-end argument," a system design principle introduced by Saltzer et al. [1984]. This principle states that, rather often, functions such as reliable file transfer can be completely and correctly implemented only with the knowledge and help of the application standing at the endpoints of the underlying system (e.g., the communication network).

This does not mean that everything should be done at application level; fault tolerance strategies in the underlying hardware and operating system can have a strong impact on system performance. However, an extraordinarily reliable communication system, guaranteeing that no packet is lost, duplicated, corrupted, or delivered to the wrong addressee, does not reduce the burden on the application program of ensuring reliability: For instance, for reliable file transfer, the application programs that perform the transfer must still supply a file-transfer-specific end-to-end reliability guarantee.

Hence one can conclude that:

> Pure hardware-based or operating system-based solutions to fault-tolerance, though often characterized by a higher degree of transparency, are not fully capable of providing complete end-to-end tolerance to faults in the user application. Furthermore, relying solely on the hardware and the operating system develops only partially satisfying solutions; requires a large amount of extra resources and costs; and is often characterised by poor service portability [Saltzer et al. 1984; Siewiorek and Swarz 1992].

[4]As Leslie Lamport effectively summarized, "a distributed system is one in which I cannot get something done because a machine I've never heard of is down."

[5]The Ariane 5 program reused the extensively tested software used in Ariane 4. As such, the software had been thoroughly tested and complied to Ariane 4 specifications. Unfortunately, with the specifications for Ariane 5 were different. A dormant design fault had never been unraveled, simply because the operating conditions of Ariane 4 were different from those of Ariane 5. This failure entailed a loss of about 370 million euros [Le Lann 1996].

2.3. Strategies, Problems, and Key Properties

The aforesaid conclusions justify the strong need for ALFT; as a consequence of this need, several approaches to ALFT have been devised during the last three decades (see Section 3 for a brief survey). Such a long research period hints at the complexity of the design problems underlying ALFT engineering, which include:

(1) how to incorporate fault tolerance in the application layer of a computer program;

(2) which fault-tolerance provisions to support; and

(3) how to manage the fault-tolerant code.

Problem 1 is also known as the problem of the *system-structure-to-software fault tolerance*, first proposed by Randell [1975]. It states the need of appropriate structuring techniques such that the incorporation of a set of fault-tolerance provisions in the application software might be performed in a simple, coherent, and well-structured way. Indeed, poor solutions to this problem result in a huge degree of *code intrusion*: In such cases, the application code addressing functional requirements and that addressing fault-tolerance requirements are intermixed into one large and complex application software. We next give three consequent obstacles to software fault-tolerance design.

—Amalgamating these two types of application code greatly complicates the task of the developer and requires expertise in both the application domain and in fault tolerance. Negative repercussions on the development times and costs are to be expected.

—The maintenance of the resulting code, both for the functional part and for the fault-tolerance provisions, is more complex, costly, and error prone.

—Furthermore, the overall complexity of the software product is increased, which is detrimental to its resilience to faults.

One can conclude that, with respect to the first problem, an ideal system structure should guarantee an adequate *Separation between the functional and fault-tolerance Concerns* (SC).

Moreover, the design choice of which fault-tolerance provisions to support can be conditioned by the adequacy of the syntactical structure at "hosting" the various provisions. The well-known quotation by B. L. Whorf succinctly captures this concept:

> Language shapes the way we think, and determines what we can think about.

Indeed, as explained in Section 2.3.1, a nonoptimal answer to problem 2 may:

—require a high degree of redundancy, and

—rapidly consume large amounts of the available redundancy,

thus increasing the costs and reducing reliability. One can conclude that devising a syntactical structure offering straightforward support to a large set of fault-tolerance provisions can be an important aspect of an ideal system structure for ALFT. In the following, this property will be called *syntactical adequacy* (SA).

Finally, one can observe that another important aspect of an ALFT architecture is the way the fault-tolerant code is managed, at compile time as well as at runtime. Evidence for this statement can be found by observing that a number of important choices pertaining to the adopted fault-tolerance provisions, such as the parameters of a temporal redundancy strategy, are a consequence of an analysis of the environment in which the application is to be deployed and run.[6] In other words, depending on the

[6]For instance, if an application is to be moved from a domestic environment to another one characterized by a higher electro-magnetic interference (EMI), it is reasonable to assume that, for example, the number of replicas of some protected resource should be increased accordingly.

target environment, the set of (external) impairments that might affect the application can vary considerably. Now, while it may be in principle straightforward to port an existing code to another computer system, *porting the service* supplied by that code may require a proper adjustment of the aforementioned choices, namely the parameters of the adopted provisions [De Florio and Blondia 2007a]. Effective support towards managing the parametrization of the fault-tolerant code, and of its maintenance in general, could guarantee *fault-tolerance software reuse*. Therefore, offline and online (dynamic) management of fault-tolerance provisions and their parameters may be an important requirement for any satisfactory solution of problem 3. As further motivated in Section 2.3.1, ideally the fault-tolerant code should *adapt* itself to the current environment. Furthermore, any satisfactory management approach should not overly increase the complexity of the application, as this would be detrimental to dependability. Let us call this property *adaptability* (A).

Let us refer collectively to properties SC, SA, and A as the *structural attributes* of ALFT.

The various approaches to ALFT surveyed in Section 3 provide different system structures to solve the aforementioned problems. Three structural attributes are used in that section in order to provide a qualitative assessment with respect to various application requirements. These structural attributes constitute, in a sense, a base with which to perform this assessment. One of the major conclusions of the survey therein is that none of the surveyed approaches is capable of providing the best combination of values of the three structural attributes in every application domain. For specific domains such as object-oriented distributed applications, satisfactory solutions have been devised, at least for SC and SA Nonethless, only partial solutions exist, for instance, when dealing with the class of distributed or parallel applications not based on the object model.

The aforesaid obstruction has, as a matter of fact, been efficaciously captured by Lyu, who calls this situation "the software bottleneck" of system development [Lyu 1998b]: In other words, there is evidence of an urgent need for systematic approaches to assure software reliability within a system [Lyu 1998b] while effectively addressing the previously described problems. In the cited paper, Lyu remarks how "developing the required techniques for software reliability engineering is a major challenge to computer engineers, software engineers and engineers of related disciplines."

2.3.1. Fault-Tolerance, Redundancy, and Complexity. A well-known result by Shannon [1993] tells us that from any unreliable channel, it is possible to set-up a more reliable one by increasing the degree of information redundancy. This means that it is possible to tradeoff reliability and redundancy of a channel. The authors observe that the same can be said for a fault-tolerant system because fault tolerance is, in general, the result of some strategy effectively exploiting some form of redundancy: time-, information-, and/or hardware redundancy [Johnson 1989]. This redundancy has a cost penalty attached, however. Its ability to address a weak failure semantics and to span many failure behaviors effectively translates into higher reliability. Regardless, redundancy:

(1) requires large amounts of extra resources, and therefore implies a high cost penalty; and
(2) consumes large amounts of extra resources, which translates into rapid exhaustion of the extra resources.

For instance, Lamport et al. [1982] set the minimum level of redundancy required for tolerating Byzantine failures to a value greater than that required for tolerating, for example, value failures. Using the simplest of the algorithms described in the cited

paper, a 4-modular-redundant (4-MR) system can only withstand a single Byzantine failure, while the same system may exploit its redundancy to withstand up to three crash faults (though no other kind of fault) [Powell 1997]. In other words:

> After the occurrence of a crash fault, a 4-MR system with strict Byzantine failure semantics has exhausted its redundancy and is no more dependable than a no[nr]edundant system supplying the same service, while the crash failure semantics system is able to survive to the occurrence of that and two other crash faults. On the other hand, the latter system, subject to just one Byzantine fault, would fail regardless its redundancy.

We can conclude that for any given level of redundancy, trading the complexity of the failure mode against the number and types of faults tolerated may be an important capability for an effective fault-tolerant structure. Dynamic adaptability to different environmental conditions[7] may provide a satisfactory answer to this need, especially when this additional complexity does not burden (and hence jeopardize) the application. Ideally, such complexity should be part of a custom architecture and not part of the application. On the other hand, an embedding in the application of complex failure semantics covering many failure modes implicitly promotes complexity, as it may require the implementation of many recovery mechanisms. This complexity is detrimental to the dependability of the system, as it is in itself a significant source of design faults. Furthermore, the isolation of this complexity outside the user application may allow cost-effective verification, validation, and testing. These processes may be unfeasible at the application level.

The authors conjecture that a satisfactory solution to the design problem of management of the fault-tolerant code (presented in Section 2.3) may translate into an optimal management of the failure semantics (with respect to the involved penalties). In other words, we conjecture that linguistic structures characterized by high *adaptability* (A) may be better suited to cope with the preceding problems.

3. CURRENT APPROACHES TO APPLICATION-LEVEL FAULT TOLERANCE

One of the conclusions drawn in Section 1 is that for the system to be made fault tolerant, we must also include provisions for fault tolerance in the application layer of a computer program. In this context, the problem of which system structure to use for ALFT has been proposed. This section provides a critical survey of the state-of-the-art on embedding fault-tolerance means in the application layer.

According to the literature, at least six classes of method can be used for embedding fault-tolerance provisions in the application layer of a computer program. This section describes these approaches and points out positive and negative aspects of each with respect to the structural attributes defined in Section 2.3, as well as to various application domains. A nonexhaustive list of the systems and projects implementing these approaches is also given. Conclusions are drawn in Section 4, where the need for more effective approaches is recognized.

Two of the previously mentioned approaches derive from well-established research in software fault-tolerance; Lyu [1998b, 1996, 1995] refers to them as single-version and multiple-version software fault tolerance. They are dealt with in Section 3.1. A third approach, described in Section 3.2, is based on metaobject protocols. It is derived from the domain of object-oriented design and can also be used for embedding services other than those related to fault tolerance. A fourth approach translates into developing

[7]The following quote by J. Horning [1998] captures very well how relevant may be the role of the environment with respect to achieving the required quality of service: "What is the most often overlooked risk in software engineering? That the environment will do something the designer never anticipated."

new, custom high-level distributed programming languages or enhancing preexistent languages of that kind. It is described in Section 3.3. Aspect-oriented programming as a fault-tolerance structuring technique is discussed in Section 3.4. Finally, Section 3.5 describes an approach based on a special recovery task monitoring the execution of the user task.

3.1. Single- and Multiple-Version Software Fault-Tolerance

A key requirement for the development of fault-tolerant systems is the availability of *replicated resources* in hardware or software. A fundamental method employed to attain fault tolerance is *multiple computation*, namely, N-fold ($N \geq 2$) replications in the three domains next given.

—*Time*. This refers to repetition of computations.

—*Space*. This is replication as the adoption of multiple hardware channels (also called "lanes").

—*Information*. This means the adoption of multiple versions of software.

Following Avižienis [1985], it is possible to characterize at least some of the approaches towards fault tolerance by means of a notation resembling the one used to classify queueing systems models [Kleinrock 1975]. Specifically,

$$n\mathrm{T}/m\mathrm{H}/p\mathrm{S},$$

the meaning of which is "n executions, on m hardware channels, of p programs." The nonfault-tolerant system, or 1T/1H/1S, is called a *simplex* in the cited paper.

3.1.1. Single-Version Software Fault-Tolerance. Single-version software fault-tolerance (SV) is basically the embedding into the user application of a simplex system of error detection or recovery features, for example, atomic actions [Jalote and Campbell 1985], checkpoint-and-rollback [Elnozahy et al. 2002], or exception handling [Cristian 1995]. The adoption of SV in the application layer requires the designer to concentrate, in one physical location (i.e., the source code of the application), both the specification of what to do in order to perform some user computation and the strategy such that faults are tolerated when they occur. As a result, the size of the problem addressed is increased. *A fortiori*, this translates into an increase in size of the user application, which induces loss of transparency, maintainability, and portability while increasing development times and costs.

A partial solution to this loss in portability and these higher costs is given by the development of libraries and frameworks created under strict software engineering processes. In the following, two examples of this approach are presented: the EFTOS library and the SwiFT system.

—*The EFTOS Library.* EFTOS [De Florio et al. 1998a] (i.e., "Embedded, Fault-Tolerant Supercomputing") is the name of ESPRIT-IV project 21012. The aims of this project were to investigate approaches to add fault tolerance to embedded high-performance applications in a flexible and cost-effective way. The EFTOS library was first implemented on a Parsytec CC system [Parsytec 1996], a distributed-memory MIMD supercomputer consisting of processing nodes based on PowerPC 604 microprocessors at 133MHz, dedicated high-speed links, I/O modules, and routers.

Through adoption of the EFTOS library, the target embedded parallel application is plugged into a hierarchical, layered system whose structure and basic components (depicted in Figure 2) are described as follows.

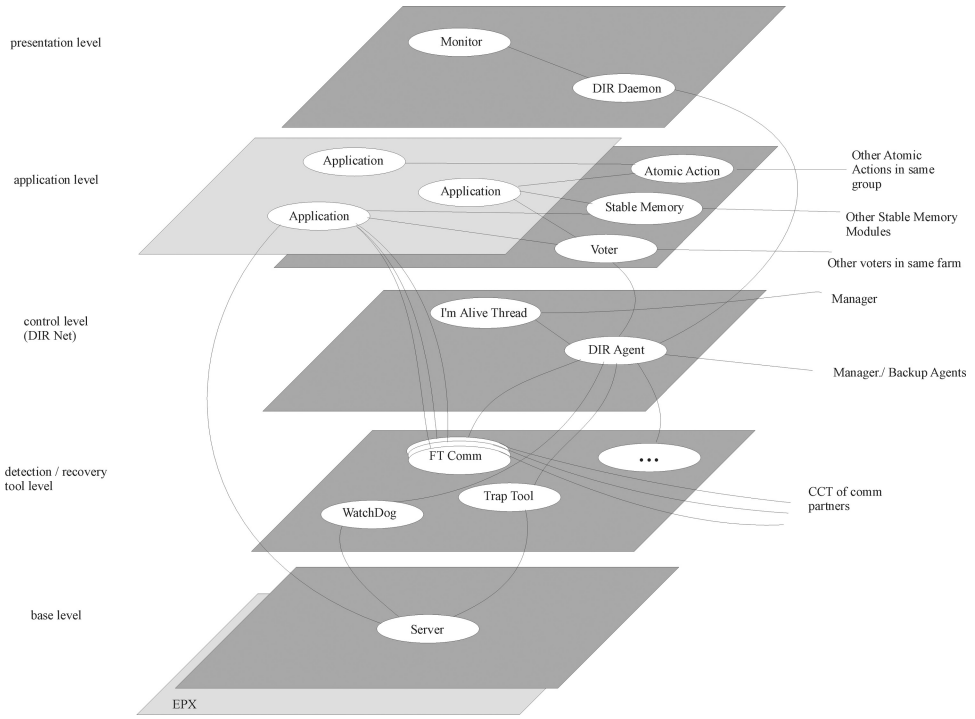

Fig. 2. The structure of the EFTOS library. Light gray has been used for the operating system and the user application, while dark gray layers pertain to EFTOS.

—At the base level, there is a distributed net of "servers," whose main task is mimicking some possibly missing (with respect to POSIX standards) operating system functionalities such as remote thread creation.

—One level upward (at the detection tool layer) there exists a set of parameterizable functions managing error detection, referred to as "Dtools." These basic components are plugged into the embedded application to make it more dependable. EFTOS supplies a number of these Dtools (e.g., a watchdog timer thread and a trap-handling mechanism) plus an API for incorporating user-defined EFTOS-compliant tools.

—At the third level (control layer), a distributed application called "DIR net" (its name stands for "detection, isolation, and recovery network") is used to coherently combine the Dtools. This is done to ensure consistent fault-tolerance strategies throughout the system, and to play the role of a backbone handling information to and from the fault-tolerance elements [De Florio et al. 2000; De Florio 1998]. The DIR net can be regarded as a fault-tolerant network of crash-failure detectors connected to other peripheral error detectors. Each node of the DIR net is "guarded" by an <I'm Alive> thread that requires the local component to periodically send "heartbeats" (signs of life). A special component, called RINT, manages error recovery by interpreting a custom language called RL [De Florio and Deconinck 2002; De Florio 1997a].

—At the fourth level (application layer), the Dtools and components of the DIR net are combined into dependable mechanisms; that is, methods to guarantee fault-tolerant communication [Efthivoulidis et al. 1998], tools implementing a virtual stable memory [De Florio et al. 2001], a distributed voting mechanism called the "voting farm" [De Florio 1997b; De Florio et al. 1998a, 1998b], and so forth.

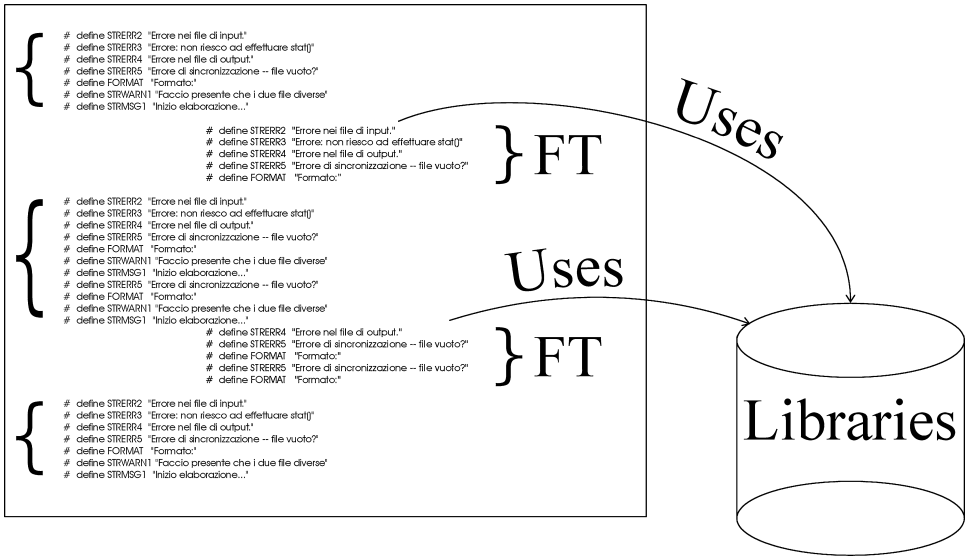

Fig. 3. A fault-tolerant program according to an SV system.

—The highest level (presentation layer) is given by a hypermedia distributed application which monitors the structure and state of the user application [De Florio et al. 1998c]. This application is based on a special CGI script [Kim 1996] called DIR Daemon, which continuously takes its inputs from the DIR net, translates them into HTML, and remotely controls a Netscape browser [Zawinski 1994] so that it renders this HTML data.

—*The SwiFT System.* SwiFT [Huang et al. 1996] stands for "software-implemented fault tolerance." It includes a set of reusable software components consisting of `watchd`, a general-purpose UNIX daemon watchdog timer; `libft`, a library of fault-tolerance methods including a single-version implementation of recovery blocks and N-version programming (see Section 3.1.2); `libckp`, which is a user-transparent checkpoint-and-rollback library; a file replication mechanism called `REPL`; and `addrejuv`, a special "reactive" feature of `watchd` [Huang et al. 1995] that allows for software rejuvenation.[8] SwiFT has been successfully used and proved an efficient and economical means to increase the level of fault tolerance in a software system where residual faults are present and their toleration is less costly than their full elimination [Lyu 1998b]. A relatively small overhead is introduced in most cases [Huang and Kintala 1995].

Conclusions. Figure 3 shows the main characteristics of the SV approach: The functional and fault-tolerant codes are intertwined and the developer has to deal with the two concerns at the same time, even with the help of libraries of fault-tolerance provisions. In other words, SV requires the application developer to be an expert in fault tolerance as well, because he (she) has to integrate in the application a number of fault-tolerance provisions among those available in a set of ready-made basic tools. His (her) responsibility is to do so in a coherent, effective, and efficient way. As observed in

[8]Software rejuvenation [Huang et al. 1995; Bao et al. 2003] offers tools for periodical and graceful termination of an application with immediate restart, so that possible erroneous internal states due to transient faults are wiped out before they cause a failure.

Section 2.3, the resulting code is a mixture of functional- and custom error-management code that does not always offer an acceptable degree of portability and maintainability. The functional and nonfunctional design concerns are not kept apart from SV, hence one can conclude that (qualitatively) SV exhibits poor separation of concerns (SC). This in general has a bad impact on design and maintenance costs.

As for syntactical adequacy (SA), we observe that, following SV, the fault-tolerance provisions are offered to the user through an interface based on a general-purpose language such as C or C++. As a consequence, very limited SA can be achieved by SV as a system structure for ALFT.

Furthermore, no support is provided for offline and online configuration of the fault-tolerance provisions. Consequently, we regard the adaptability (A) of this approach as insufficient.

On the other hand, tools in SV libraries and systems give the user the ability to deal with fault-tolerance "atoms" without having to worry about their actual implementation. Moreover, these tools provide a good ratio of cost over improvement of the dependability attributes, sometimes introducing a relatively small overhead. Using these tool sets the designer can reuse existing, long-tested, and sophisticated pieces of software without having each time to "reinvent the wheel."

Finally, it is important to remark that, in principle, SV poses no restrictions on the class of applications that may be tackled with it.

3.1.2. Multiple-Version Software Fault-Tolerance. This section describes multiple-version software fault-tolerance (MV), an approach which requires N ($N \geq 2$) independently designed versions of software. MV systems are therefore xT/yH/NS systems. In MV, the same service or functionality is supplied by N pieces of code that have been designed and developed by different independent software teams.[9] The aim of this approach is to reduce the effects of design faults due to human mistakes committed at design time. The most frequently used configurations are NT/1H/NS; that is, N sequentially applicable alternate programs using the same hardware channel, and 1T/NH/NS, which is based on the parallel execution of alternate programs on N (possibly diverse) hardware channels.

Two major approaches exist: The first is known as a recovery block [Randell 1975; Randell and Xu 1995], and is dealt with next. Then, the second approach, the so-called N-version programming [Avižienis 1995, 1985], is described.

—*The Recovery Blocks Technique.* Recovery blocks are usually implemented as NT/1H/NS systems. The technique addresses residual software design faults. It aims at providing fault-tolerant functional components which may be nested within a sequential program. Other versions of the approach, implemented as 1T/NH/NS systems, are suited for parallel or distributed programs [Scott et al. 1985; Randell and Xu 1995].

The recovery blocks technique is similar to the hardware fault-tolerance approach known as "standby sparing," which is described, for example, in Johnson [1989]. The approach is summarized in Figure 4: On entry to a recovery block, the current state of the system is checkpointed. A primary alternate is executed. When it ends, an acceptance test checks whether the primary alternate successfully accomplished its objectives. If it has not, a backward recovery step reverts the system state back to its original value and a secondary alternate takes over the task of the primary alternate. When the secondary

[9]This requirement is well explained by Randell [1975]: "All fault-tolerance must be based on the provision of useful redundancy, both for error detection and error recovery. In software the redundancy required is not simple replication of programs but *redundancy of design*." Footnote 5 briefly reports on the consequences of a well-known case of redundant design.

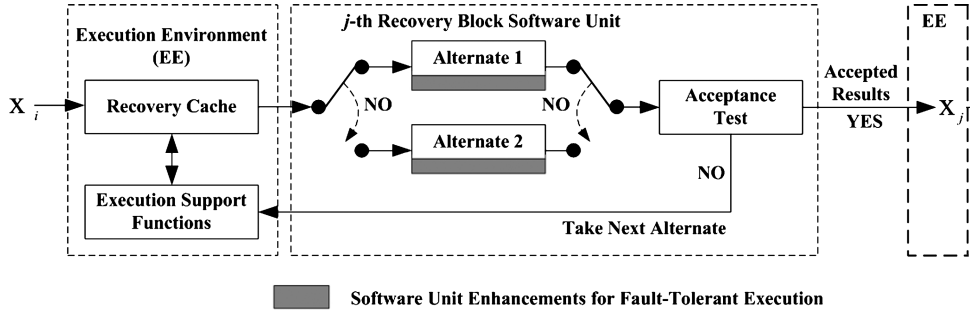

Fig. 4. The recovery block model with two alternates. The execution environment is charged with the management of the recovery cache and execution support functions (used to restore the state of the application when the acceptance test is not passed), while the user is responsible for supplying both alternates and the acceptance test.

alternate ends, the acceptance test is executed again. The strategy goes on until either an alternate fulfils its tasks or all alternates are executed without success. In such a case, an error routine is executed. Recovery blocks can be nested: In this case the error routine invokes recovery in the enclosing block [Randell and Xu 1995]. An exception triggered within an alternate is managed as a failed acceptance test. A possible syntax for recovery blocks is as follows.

```
ensure       acceptance test
by           primary alternate
else by      alternate 2
     .
     .
else by      alternate N
else error
```

Note how this syntax does not explicitly show the recovery step that should be carried out transparently by a runtime executive.

The effectiveness of recovery blocks rests to a great extent on the coverage of the error detection mechanisms adopted, the most crucial component of which is the acceptance test. A failure of the acceptance test is a failure of the whole recovery blocks strategy. For this reason, the acceptance test must be simple, must not introduce huge runtime overheads, must not retain data locally, and so forth. It must be regarded as the ultimate means for detecting errors, though not an exclusive one. Assertions and runtime checks, possibly supported by underlying layers, need to buttress the strategy and reduce the probability of an acceptance test failure. Another possible failure condition for the recovery blocks approach is given by an alternate failing to terminate. This may be detected by a time-out mechanism that could be added to the recovery blocks. This addition obviously further increases the complexity.

The SwiFT library described in Section 3.1.1 implements recovery blocks in the C language as follows.

```
#include <ftmacros.h>
...
ENSURE(acceptance-test) {
          primary alternate;
```

```
} ELSEBY {
   alternate 2;
} ELSEBY {
   alternate 3;
}
...
ENSURE;
```

Unfortunately, this approach does not cover any of the aforementioned requirements for enhancing the error detection coverage of the acceptance test. This would clearly require a runtime executive that is not part of this strategy. Other solutions, based on enhancing the grammar of preexisting programming languages such as Pascal [Shrivastava 1978] and Coral [Anderson et al. 1985], have some impact on portability. In both cases, code intrusion is not avoided. This translates into difficulties when trying to modify or maintain the application program without interfering much with the recovery structure, and vice-versa: as when trying to modify or maintain the recovery structure without interfering much with the application program. Hence, one can conclude that recovery blocks are characterized by unsatisfactory values of the structural attribute SC. Furthermore, a system structure for ALFT based exclusively on recovery blocks does not satisfy attribute SA.[10] Finally, regarding attribute A, one can observe that recovery blocks comprise a rigid strategy that does not allow offline configuration nor (*a fortiori*) code adaptability.

On the other hand, recovery blocks have been successfully adopted throughout the last 30 years in many different application fields. It has been successfully validated by a number of statistical experiments and through mathematical modeling [Randell and Xu 1995]. Its adoption as the sole fault-tolerance means while developing complex applications has resulted, in some cases [Anderson et al. 1985], in a failure coverage of over 70%, with acceptable overheads in memory space and CPU time.

A negative aspect in any MV system is given by development and maintenance *costs* that grow as a monotonic function of x, y, z in any xT/yH/zS system.

—**N**-*Version Programming.* N-version programming (NVP) systems are built from generic architectures based on redundancy and consensus. Such systems usually belong to class 1T/NH/NS, and less often to class NT/1H/NS. Specifically, NVP is defined by its author Avižienis [1985] as "the independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification." These N programs, called versions, are developed to be executed in parallel. This system constitutes a fault-tolerant software unit that depends on a generic decision algorithm to determine a consensus or majority result from the individual outputs of two or more versions of the unit.

Such a strategy (depicted in Figure 5) has been developed under the fundamental conjecture that independent designs translate into random component failures, namely, statistical independence. Such a result would guarantee that correlated failures do not translate into immediate exhaustion of the available redundancy, as would happen, for example, using N copies of the same version. Replicating software would also mean replicating any dormant software fault in the source version; see, for example, the accidents with the Therac-25 linear accelerator [Leveson 1995] or Ariane 5 flight 501 [Inquiry Board Report 1996]. According to Avižienis, independent generation of the versions significantly reduces the probability of correlated failures. Unfortunately,

[10]Randell himself states that, given the ever-increasing complexity of modern computing, there is still an urgent need for "richer forms of structuring for error recovery and for design diversity" [Randell and Xu 1995].

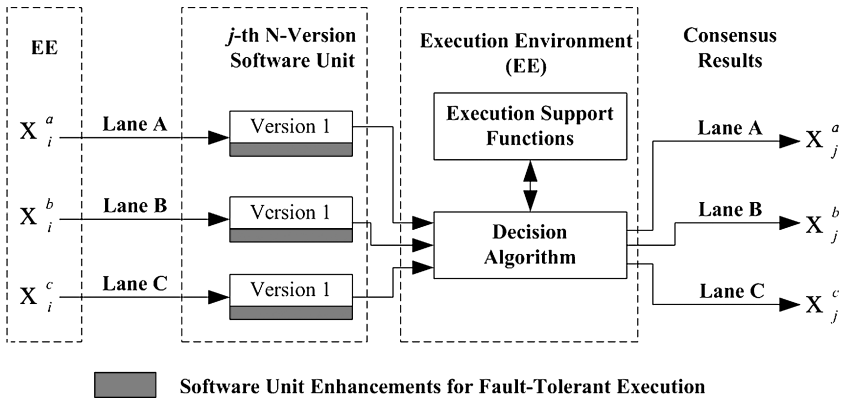

Fig. 5. The N-version software model when $N = 3$. The execution environment is charged with the management of the decision algorithm and with the execution support functions. The user is responsible for supplying the N versions. Note how the decision algorithm box takes care also of multiplexing its output onto the three hardware channels, also called "lanes."

a number of experiments [Eckhardt et al. 1991] and theoretical studies [Eckhardt and Lee 1985] have shown that this assumption is not always correct.

The main differences between recovery blocks and NVP are next described.

—Recovery blocks (in original form) constitutes a sequential strategy, whereas NVP allows concurrent execution.

—Recovery blocks require the user to provide a fault-free, application-specific, effective acceptance test, while NVP adopts a generic consensus or majority voting algorithm that can be provided by the execution environment (EE).

—Recovery blocks allow different correct outputs from the alternates, while the general-purpose character of the consensus algorithm of NVP calls for a single correct output.[11]

The two models collapse when the acceptance test of recovery blocks is done as in NVP; that is, when the acceptance test is a consensus on the basis of the outputs of the different alternates.

Conclusions. As with recovery blocks, NVP has been successfully adopted for many years in various application fields, including safety-critical airborne and spaceborne applications. The generic NVP architecture, based on redundancy and consensus, addresses parallel and distributed applications written in any programming paradigm. A generic, parameterizable architecture for real-time systems that supports the NVP strategy straightforwardly is GUARDS [Powell et al. 1999].

It is worth remarking that the EE (also known as N-version executive) is a complex component that needs to manage a number of basic functions; for instance, execution of the decision algorithm, the assurance of input consistency for all versions,

[11]This weakness of NVP can be narrowed, if not solved, adopting the approach used in the so-called "voting farm" [De Florio et al. 1998a, 1998b; De Florio 1997b]: a generic voting tool designed by one of the authors of this article within the framework of his participation in project EFTOS (see Section 3.1.1). Specifically, such a tool works with opaque objects that are compared by means of a user-defined function. This function returns an integer value representing a "distance" between any two objects to be voted. The user may choose between a set of predefined distance functions or to develop an application-specific distance function. Doing the latter, a distance may be endowed with the ability to assess that bitwise different objects are semantically equivalent. Of course, the user is still responsible for supplying a bug-free distance function, although assisted in this simpler task by a number of template functions supplied with this tool.

interversion communication, version synchronization, and the enforcement of timing constraints [Avižienis 1995]. On the other hand, this complexity is not part of the application software (the N versions) and does not need to be aware of the fault-tolerance strategy. An excellent degree of transparency can be reached, thus guaranteeing a good value for attribute SC. Furthermore, as mentioned in Section 2.3, the cost and time required by a thorough verification, validation, and testing of this architectural complexity may be acceptable, while charging them to each application component is certainly not a cost-effective option.

Regarding attribute SA, the same considerations provided when describing recovery blocks hold for NVP: Also in this case a single fault-tolerance strategy is followed. For this reason we assess NVP as unsatisfactory regarding attribute SA.

Offline adaptability to "bad" environments may be reached by increasing the value of N, though this requires developing new versions: a costly activity in terms of both time and cost. Furthermore, the architecture does not allow any dynamic management of the fault-tolerance provisions. We conclude that attribute A is poorly addressed by NVP.

Portability is restricted by the portability of the EE and of each of the N versions. Maintainability actions may also be problematic: They need to be replicated and validated N times, as well as performed according to the NVP paradigm, so as not to impact negatively on the statistical independence of failures. Clearly, the same considerations apply to recovery blocks as well. In other words, the adoption of multiple-version software fault-tolerance provisions always implies a penalty on maintainability and portability.

Limited NVP support has been developed for "conventional" programming languages such as C. For instance, `libft` (see Section 3.1.1) implements NVP as follows.

```
#include <ftmacros.h>
...
NVP
VERSION{
          block 1;
          SENDVOTE(v_pointer, v_size);
}
VERSION{
          block 2;
          SENDVOTE(v_pointer, v_size);
}
...
ENDVERSION(timeout, v_size);
if (!agreeon(v_pointer)) error_handler();
ENDNVP;
```

Note that this particular implementation extinguishes the potential transparency that in general characterizes NVP, as it requires including some nonfunctional code. This translates into an unsatisfactory value for attribute SC. Note also that the execution of each block is, in this case, carried out sequentially.

It is important to note how the adoption of NVP as a system structure for ALFT requires a substantial increase in development and maintenance costs: Both the 1T/NH/NS and NT/1H/NS systems have a cost function growing quadratically with N. The author of the NVP strategy remarks how such costs are repaid by the gain in trustworthiness. This is certainly true when dealing with systems possibly subjected to catastrophic failures let us recall once more the case of the Ariane 5 flight 501 [Inquiry

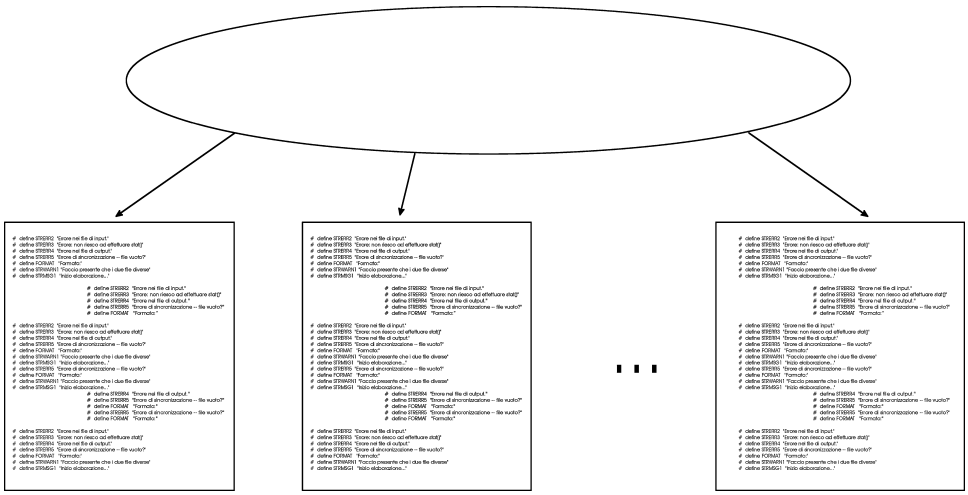

Fig. 6. A fault-tolerant program according to an MV system.

Board Report 1996]. Clearly, risk of rapid exhaustion of redundancy (due to a burst of correlated failures) caused by a single or few design faults is serious. It justifies and calls for the adoption of other fault-tolerance provisions within and around the NVP unit in order to deal with the case of its failure..

Figure 6 shows the main characteristics of the MV approach: Several replicas (portions) of the functional code are produced and managed by a control component. In recovery blocks this component is often coded side-by-side with the functional code, while in NVP this is usually a custom hardware box.

3.1.3. A Hybrid Case: Data Diversity. A special hybrid case is given by data diversity [Ammann and Knight 1988]. A data diversity system is a 1T/NH/1S (less often a NT/1H/1S). It can be concisely described as an NVP system in which N equal replicas are used as versions, but each replica receives a different minor perturbation of the input data. Under the hypothesis that the function computed by the replicas is non-chaotic, that is, does not produce very different output values when fed slightly different inputs, data diversity may be a cost-effective way to fault tolerance. Clearly in this case the vote, mechanism does not run a simple majority vote but some vote fusion algorithm [Lorczak et al. 1989]. A typical application of data diversity is that of a real-time control program, where sensor resampling or a minor perturbation in the sampled sensor value may be able to prevent a failure. Being substantially an NVP system, data diversity reaches the same values for the structural attributes. The greatest advantage of this technique is its drastically decreased design and maintenance costs, since design diversity is avoided.

3.2. Metaobject Protocols and Reflection

Some of the negative aspects pointed out while describing single- and multiple-version software approaches can be, in some cases, weakened (if not solved) by means of a generic structuring technique. This method allows one to reach in some cases an adequate degree of flexibility, transparency, and separation of design concerns: the adoption of metaobject protocols (MOPs) [Kiczales et al. 1991]. The idea is to "open" the implementation of the runtime executive of an object-oriented language (e.g., C++ or Java) so

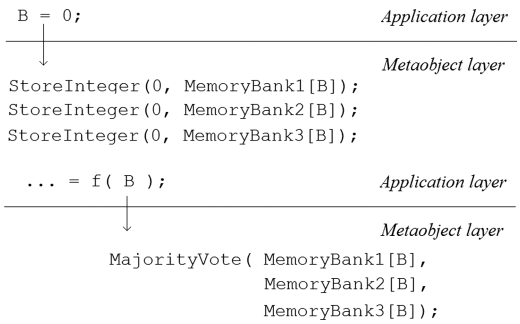

Fig. 7. A MOP may be used to realize, for example, triple-redundant memories in a fully transparent way.

that the developer can adopt and program different, custom semantics, adjusting the language to the needs of the user and environment. Using MOPs, the programmer can modify the behavior of fundamental features such as methods invocation, object creation and destruction, and member access. The transparent management of spatial and temporal redundancy [Taylor et al. 1980] is a case where MOPs seem particularly adequate: For instance, an MOP programmer may easily create "triple-redundant" memory cells to protect his (her) variables against transient faults, as depicted in Figure 7.

The key concept behind MOPs is that of computational reflection, or the causal connection between a system and a metalevel description representing the structural and computational aspects of that system [Maes 1987]. MOPs offer the metalevel programmer a representation of a system as a set of metaobjects. Metaobjects are those objects that represent and reflect properties of "real" objects, namely, those objects that constitute the functional part of the user application. Metaobjects can, for instance, represent the structure of a class, object interaction, or code of an operation. This mapping process is called reification [Robben 1999].

The causality relation of MOPs could also be extended to allow for a dynamic reorganization of the structure and operation of a system; for instance, to perform reconfiguration and error recovery. The basic object-oriented feature of inheritance can be used to enhance the reusability of the FT mechanisms developed with this approach.

3.2.1. Project FRIENDS. An architecture supporting this approach is the one developed in the framework of project FRIENDS [Fabre and Pérennou 1998, 1996]. The name FRIENDS stands for "flexible and reusable implementation environment for your next dependable system." This project aims at implementing a number of fault-tolerance provisions (e.g., replication, group-based communication, synchronization, voting, etc. [van Achteren 1997]) at metalevel. In FRIENDS a distributed application is a set of objects interacting via the proxy model, a proxy being a local intermediary between each object and any other (possibly replicated) object. FRIENDS uses the metaobject protocol provided by Open C++, a C++ preprocessor that provides control over instance creation and deletion, state access, and invocation of methods.

Other ALFT architectures exploiting the concept of metaobject protocols within custom programming languages are reported in Section 3.3.

Conclusions. MOPs are indeed a promising system structure for embedding different nonfunctional concerns in the application level of a computer program. MOPs work at *language* level, providing a means to modify the semantics of basic object-oriented language building blocks, such as object creation and deletion, calling and termination of class methods, and so forth. This appears to match perfectly with a proper subset of

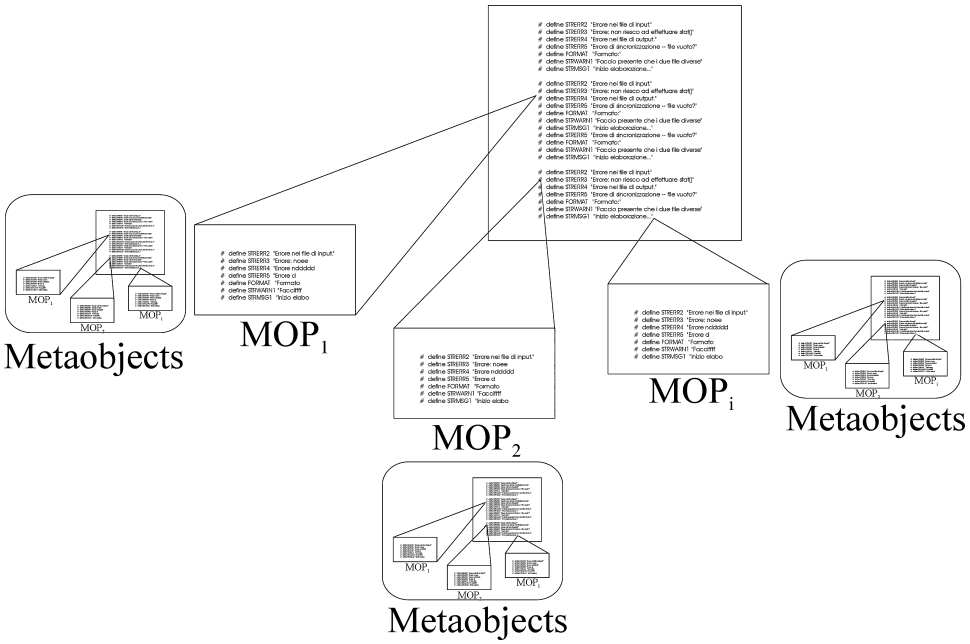

Fig. 8. A fault-tolerant program according to a MOP system.

the possible fault-tolerance provisions, especially those such as transparent object redundancy that can be straightforwardly managed with the metaobject approach. When dealing with these fault-tolerance provisions, MOPs provide a perfect separation of the design concerns, namely, optimal SC. Some other techniques, specifically those which might be described as the most coarse-grained (e.g., distributed recovery blocks [Kim and Welch 1989]), appear to be less suited to efficient implementation via MOPs. These techniques work at a distributed, macroscopic level.

Figure 8 synthesizes the main characteristics of the MOP approach: The fault-tolerance programmer defines a number of metaobject protocols and associates them with method invocations or other grammar cases. Each time the functional program enters a certain grammar case, the corresponding protocol is transparently executed. Each protocol has access to a representation of the system through its metaobjects, by means of which it can also perform actions on the corresponding "real" objects.

MOPs appear to constitute a promising technique for a transparent, coherent, and effective adoption of some existing FT mechanisms and techniques. A number of studies confirm that MOPs reach efficiency in some cases [Kiczales et al. 1991; Masuhara et al. 1992], though no experimental or analytical evidence thus far allows to estimate the practicality and generality of this approach: "[w]hat reflective capabilities are needed for what form of fault-tolerance, and to what extent these capabilities can be provided in more-or-less conventional programming languages, and allied to other structuring techniques [e.g., recovery blocks or NVP] remain to be determined" [Randell and Xu 1995]. In other words, it is still an open question as to whether MOPs represent a practical solution towards effective integration of most of the existing fault-tolerance mechanisms in user applications.

The aforesaid situation reminds the authors of another, regarding the "quest" for a novel computational paradigm for parallel processing which would be capable of dealing effectively with the widest class of problems, as the von Neumann paradigm does for

sequential processing, though with the highest degree of efficiency and least amount of change in the original (sequential) user code. In that context, the concept of computational *grain* came up; some techniques were inherently looking at the problem "with coarse-grained glasses," that is, at macroscopic level, while others were considering the problem exclusively at microscopical level. One can conclude that MOPs offer an elegant system structure to embed a set of nonfunctional services (including fault-tolerance provisions) in an object-oriented program. It is still unclear whether this set is general enough to host, efficaciously, many forms of fault tolerance, as remarked, for instance, in Randell and Xu [1995] and Lippert and Videira Lopes [2000]. It is therefore difficult to establish a qualitative assessment of attribute SA for MOPs.

The runtime management MOP libraries may be used to reach satisfactory values for attribute A. To the best of the authors' knowledge, this feature is not present in any language supporting MOPs.

As evident, the target application domain is the one of object-oriented applications written with languages extended with a MOP, such as Open C++.

As a final remark, we observe how the cost of MOP-compliant fault-tolerant software design should include those related to acquisition of the extra competence and experience in MOP design tools, reification, and custom programming languages.

3.3. Enhancing or Developing Fault-Tolerance Languages

Another approach is given by working at the language level, enhancing a preexisting programming language or developing an ad hoc distributed programming language so that it hosts specific fault-tolerance provisions. The following two sections cover these topics.

3.3.1. Enhancing Preexisting Programming Languages. Enhancing a preexisting programming language means augmenting the grammar of a widespread language such as C or Pascal so that it directly supports features that can be used to enhance the dependability of its programs (e.g., recovery blocks [Shrivastava 1978]).

In the following, four classes of system based on this approach are presented: Arjuna, Sina, Linda, and FT-SR. All of them constitute provisions to develop distributed fault-tolerant systems.

—*The Arjuna Distributed Programming System.* Arjuna is an object-oriented system for portable distributed programming in C++ [Parrington 1990; Shrivastava 1995]. It can be considered as a clever blending of useful and widespread tools, techniques, and ideas; as such, it is a good example of the evolutionary approach towards application-level software fault-tolerance. It exploits remote procedure calls [Birrell and Nelson 1984] and UNIX daemons. On each node of the system, an object server connects client objects to objects supplying services. The object server also takes care of spawning objects when they are not yet running (in this case they are referred to as "passive objects"). Arjuna also exploits a "naming service," by means of which client objects request a service "by name." This transparency effectively supports object migration and replication.

As in other systems, Arjuna makes use of stub generation to specify remote procedure calls and remote manipulation of objects. A nice feature of this system is that the stubs are derived automatically from the C++ header files of the application, which avoids the need of a custom interface description language.

Arjuna offers the programmer means for dealing with atomic actions (via the two-phase commit protocol) and persistent objects. The core class hierarchy of Arjuna appears to the programmer as follows [Parrington 1990]: StateManager LockManager User-Defined Classes Lock User-Defined Lock Classes AtomicAction AbstractRecord RecoveryRecord LockRecord and other management record types; etc.

Unfortunately, it requires programmers to explicitly deal with tools to save and restore the state, manage locks, and declare in their applications those instances of the class for managing atomic actions. As its authors relate, in many respects Arjuna asks the programmer to be aware of several complexities;[12] as such, it is prejudicial to transparency and separation of design concerns. On the other hand, its good design choices result in an effective, portable environment.

—*The SINA Extensions.* The SINA [Aksit et al. 1991] object-oriented language implements the so-called composition-filters object model, a modular extension to the object model. In SINA, each object is equipped with a set of "filters." Messages sent to any object are trapped by the filters of that object. These filters possibly manipulate the message before passing it to the object. SINA is a language for composing such filters, its authors refer to it as a "composition filter language." It also supports metalevel programming through the reification of messages. The concept of composition filters allows to implement several different behaviors corresponding to different nonfunctional concerns. SINA has been designed as an attachment to existing languages: Its first implementation, SINA/st, was for Smalltalk. It has been also implemented for C++ [Glandrup 1995], and the extended language has been called C++/CF. A preprocessor is used to translate a C++/CF source into standard C++ code.

—*Fault-Tolerant Linda Systems.* The Linda [Carriero and Gelernter 1989a, 1989b] approach adopts a special model of communication known as generative communication [Gelernter 1985]. According to this model, communication is still carried out through messages, but these messages are neither sent to nor eventually read by one or more addressees. On the contrary, messages are included in a distributed (virtual) shared memory, called the tuple space, where every Linda process has equal read/write access rights. A tuple space is some sort of a shared relational database for storing and withdrawing special data objects, called tuples, sent by the Linda processes. Tuples are basically lists of objects identified by their contents, cardinality, and type. Two tuples match if they have the same number of objects, if the objects are pairwise equal for what concerns their types, and if the memory cells associated to the objects are bitwise equal. A Linda process inserts, reads, and withdraws tuples via blocking or nonblocking primitives. Reads can be performed supplying a template tuple: a prototype tuple consisting of constant fields and of fields that can assume any value. A process trying to access a missing tuple via a blocking primitive enters a wait state that continues until any tuple matching its template tuple is added to the tuple space. This allows processes to synchronize. When more than one tuple matches a template, the choice of which actual tuple to address is done in a nondeterministic way. Concurrent execution of processes is supported through the concept of "live data structures:" Tuples requiring the execution of one or more functions can be evaluated on different processors; in a sense, they become active, or "alive." Once the evaluation has finished, an (no longer active, or passive) output tuple is entered in the tuple space.

Parallelism is implicit in Linda; there is no explicit notion of network, number, and location of the system processors. Nontheless, Linda has been successfully employed in many different hardware architectures and applicative domains, resulting in a powerful programming tool that sometimes achieves excellent speedups without affecting portability issues. Unfortunately, the model does not cover the possibility of failures: For instance, the semantics of its primitives are not well defined in the case of a processor crash, and no fault-tolerance means are part of the model. Moreover, in

[12]The Arjuna stub generator attempts to compensate for these problems as far as it can automatically, but there are cases where assistance from the programmer is required. For example, heterogeneity is handled by converting all primitive types to a standard format understood by both caller and receiver.

its original form, Linda only offers single-op atomicity [Bakken and Schlichting 1995], that is, atomic execution for only a single tuple space operation. With single-op atomicity it is not possible to solve problems arising in two common Linda programming paradigms when faults occur: Both the distributed variable and the replicated-worker paradigms can fail [Bakken and Schlichting 1995]. As a consequence, a number of possible improvements have been investigated to support fault-tolerant parallel programming in Linda. Apart from design choices and development issues, many of them implement stability of the tuple space [Bakken and Schlichting 1995; Xu and Liskov 1989; Patterson et al. 1993] (via replicated state machines [Schneider 1990] kept consistent via ordered atomic multicast [Birman et al. 1991]). Others aim at combining multiple tuple-space operations into atomic transactions [Bakken and Schlichting 1995; Anderson and Shasha 1991; Cannon and Dunn 1992]. Additional techniques have also been used, such as tuple space checkpoint-and-rollback [Kambhatla 1991]. The authors also proposed an augmented Linda model for solving inconsistencies related to failures occurring in a replicated-worker environment and an algorithm for implementing a resilient replicated-worker strategy for message-passing farmer-worker applications. This algorithm can mask failures affecting a proper subset of the set of workers [De Florio et al. 1999].

Linda can be described as an extension to an existing programming language. The greater part of these Linda extensions require a preprocessor translating the extension in the host language. This is the case, for example, for FT-Linda [Bakken and Schlichting 1995], PvmLinda [De Florio et al. 1994], C-Linda [Berndt 1989], and MOM [Anderson and Shasha 1991]. A counterexample is the POSYBL system [Schoinas 1991], which implements Linda primitives with remote procedure calls, and requires the user to supply the ancillary information for distinguishing tuples.

—*FT-SR.* The FT-SR [Schlichting and Thomas 1995] tool is basically an attempt to augment the SR [Andrews and Olsson 1993] distributed programming language with mechanisms to facilitate fault tolerance. FT-SR is based on the concept of fail-stop modules (FSMs). An FSM is defined as an abstract unit of encapsulation. It consists of a number of threads that export a number of operations to other FSMs. The execution of operations is atomic. FSM can be composed so as to give rise to complex FSMs. For instance, it is possible to replicate a module $n > 1$ times and set-up a complex FSM that can survive up to $n - 1$ failures. Whenever a failure exhausts the redundancy of an FSM, be it a simple or complex FSM, a failure notification is automatically sent to a number of other FSMs so as to trigger proper recovery actions. This feature explains the name of FSM: As in fail-stop processors, either the system is correct or a notification is sent and the system stops its functions. This means that the computing model of FT-SR guarantees, to some extent, that in the absence of explicit failure notification, commands can be assumed to have been processed correctly. This greatly simplifies program development because it masks the occurrence of faults, offers guarantees that no erroneous results are produced, and encourages the design of complex, possibly dynamic failure semantics (see Section 2.3.1) based on failure notifications. Of course, this strategy is fully effective only under the hypothesis of perfect failure detection coverage, an assumption that sometimes may be found false.

Conclusions. Figure 9 synthesizes the main characteristics of the enhanced language approach: A compiler or (as in the figure) translator produces a new fault-tolerant program. In the case in the figure, the translated program belongs to class SV (see Section 3.1.1). Note how few clearly identifiable "FT" extensions are translated into larger sections of fault-tolerant code. As characteristics of an SV system, this fault-tolerant code is indistinguishable from the functional code of the application.

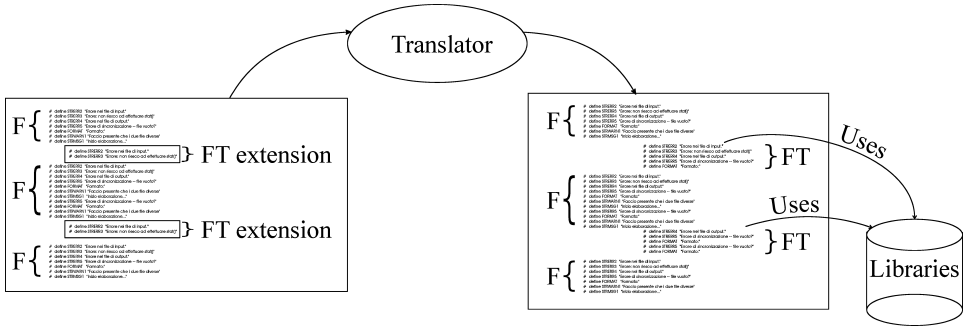

Fig. 9. A fault-tolerant program according to the enhanced language approach. Note how in this case a translator decomposes the program into an SV system.

We can conclude by stating that the approach of designing fault-tolerance enhancements for a preexisting programming language does imply an explicit code intrusion: The extensions are designed with the explicit purpose of hosting a number of fault-tolerance provisions within the single source code. We observe, however, that, being explicit, this code intrusion is such that the fault-tolerant code is generally easy to locate and distinguish from the functional code of the application. Hence, attribute SC may be positively assessed for systems belonging to this category. Following a similar reasoning and observing Figure 9 we can conclude that the design and maintenance costs of this approach are in general less than those characterizing SV.

By contrast, the problem of hosting an adequate structure for ALFT can be complicated by syntax constraints in the hosting language. This may prevent incorporation of a wide set of fault-tolerance provisions within the same syntactical structure. One can conclude that in this case attribute SA does not reach satisfactory values, at least for the examples considered in this section.

Enhancing a preexisting language is an *evolutionary* approach: In so doing, portability problems are weakened, especially when the extended grammar is translated into the plain grammar (e.g., via a preprocessor), and can be characterized by good execution efficiency [Anderson et al. 1985; Schlichting and Thomas 1995].

The approach is generally applicable, although the application must be written (or rewritten) using the enhanced language. Its adaptability (attribute A) is in general unsatisfactory because at runtime the fault-tolerant code is indistinguishable from the functional.

As a final observation, we remark that the four cases dealt with in Section 3.3.1 all stem from the domain of distributed/concurrent programming, which shows the important link between fault-tolerance issues and distributed computing.

3.3.2. Developing Novel Fault-Tolerance Programming Languages. The adoption of a custom-made language specially conceived to write fault-tolerant distributed software is discussed in the rest of this subsection.

—*ARGUS*. Argus [Liskov 1988] is a distributed object-oriented programming language and operating system. Argus was designed to support application programs such as banking systems. To capture the object-oriented nature of such programs, it provides special kinds of objects, called guardians, which perform user-definable actions in response to remote requests. To solve the problems of concurrency and failures, Argus allows computations to run as atomic transactions. Argus' target application domain is that of transaction processing.

—*The Correlate Language.* The Correlate object-oriented language [Robben 1999] adopts the concept of an active object, defined as an object that has control over the synchronization of incoming requests from other objects. Objects are active in the sense that they do not process their requests immediately; they may decide to delay a request until it is accepted, that is, until a given precondition (a guard) is met. For instance, a mailbox object may refuse a new message in its buffer until an entry becomes available in it. The precondition is a function of the state of the object and the invocation parameters; it does not imply interaction with other objects and has no side effects. If a request cannot be served according to an object's precondition, it is saved into a buffer until it becomes serviceable, or until the object is destroyed. Conditions such as an overflow in the request buffer are not dealt with in Robben [1999]. If more than a single request becomes serviceable by an object, the choice is made nondeterministically. Correlate uses a communication model called "pattern-based group communication." This type of communication goes from an "advertising object" to those objects that declare their "interest" in the advertised subject. This is similar to Linda's model of generative communication, introduced in Section 3.3.1. Objects in Correlate are autonomous in the sense that they may not only react to external stimuli, but may also give rise to autonomous operations motivated by an internal goal. When invoking a method, the programmer can choose to block it until the method is fully executed (this is called synchronous interaction) or to execute it "in the background" (asynchronous interaction). Correlate supports MOPs. It has been effectively used to offer transparent support for transaction, replication, and checkpoint-and-rollback. The first implementation of Correlate consists of a translator to plain Java plus an execution environment, also written in Java.

—*Fault-Tolerance Attribute Grammars.* The system models for application-level software fault tolerance encountered so far all have their basis in an imperative language. A different approach is based on the use of functional languages. This choice translates into a program structure that allows straightforward inclusion of a means for fault tolerance, with high degrees of transparency and flexibility. Functional models that appear particularly interesting as system structures for software fault-tolerance are those based on the concept of attribute grammars [Paakki 1995]. This paragraph briefly introduces the model known as FTAG (fault-tolerant attribute grammars) [Suzuki et al. 1996], which offers the designer a large set of fault-tolerance mechanisms. A noteworthy aspect of FTAG is that its authors explicitly address the problem of providing a syntactical model for the widest possible set of fault tolerance provisions and paradigms, developing coherent abstractions of these mechanisms while maintaining the linguistic integrity of the adopted notation. In other words, optimizing the value of attribute SA is one of the design goals of FTAG.

FTAG regards a computation as a collection of pure mathematical functions known as modules. Each module has a set of input values, called inherited attributes, and of output variables, called synthesized attributes. Modules may refer to other modules. When modules do not refer to any other module, they can be performed immediately. Such modules are called primitive modules. On the other hand, nonprimitive modules require other modules to be performed first. As a consequence, an FTAG program is executed by decomposing a "root" module into its basic submodules and then applying this decomposition process recursively to each of the submodules. This process goes on until all primitive modules are encountered and executed. The execution graph is clearly a tree, called a computation tree. This approach presents many benefits: For example, as the order in which modules are decomposed is exclusively determined by attribute dependencies among submodules, a computation tree can be turned straightforwardly into a parallel process.

The linguistic structure of FTAG allows the integration of a number of useful fault-tolerance features that address the whole range of faults: design, physical, and interaction faults. One of these features is called redoing. Redoing replaces a portion of the computation tree with a new computation. This is useful, for instance, to eliminate the effects of a portion of the computation tree that has generated an incorrect result, or whose executor has crashed. It can be used to easily implement "retry blocks" and recovery blocks. This is achieved by adding ancillary modules that test whether the original module behaved consistently with its specification and if not, give rise to a redoing: a recursive call to the original module.

Another relevant feature of FTAG is its support for replication, a concept that in FTAG translates into a decomposition of a module into N identical submodules implementing the function to replicate. Such approach is known as replicated decomposition, while involved submodules are called replicas. Replicas are executed according to the usual rules of decomposition, though only one of the generated results is used as the output of the original module. Depending on the chosen fault-tolerance strategy, this output can be, for example, the first valid output or the output of a demultiplexing function (e.g., a voter). It is worth remarking that no syntactical changes are needed, only a subtle extension of the interpretation so as to allow the involved submodules to have the same set of inherited attributes and to generate a collated set of synthesized attributes.

FTAG stores its attributes in a stable object base or in primary memory, depending on their criticality: Critical attributes can then be transparently retrieved from the stable object base after a failure. Also used is object versioning, a concept that facilitates the development of checkpoint-and-rollback strategies.

FTAG provides a unified linguistic structure that effectively supports the development of fault-tolerant software. Conscious of the importance of supporting the widest possible set of fault-tolerance means, its authors report in the cited paper [Suzuki et al. 1996] how they are investigating the inclusion of other fault-tolerance features and trying to synthesize new expressive syntactical structures for FTAG, thus further improving attribute SA.

Unfortunately, widespread adoption of this valuable tool is restricted by the limited acceptance and prevalence of the functional programming paradigm outside of academia.

Conclusions. Synthesizing, in a single meaningful picture, the main characteristics of an approach that makes use of custom fault-tolerance languages is very difficult, for the design freedom translates into entities that may have few points in common. Figure 10, for instance, synthesizes the characteristics of FTAG.

Ad hoc development of a fault-tolerance programming language allows optimal values of attribute SA to be reached in some cases. The explicit, controlled intrusion of fault-tolerant code straightforwardly encourages the adoption of high-level fault-tolerance provisions and requires dependability-aware design processes. This translates into a positive assessment for attribute SC. On the other hand, with the same reasoning of Section 3.3.1, attribute A can be generally assessed as unsatisfactory.[13]

The target application domain for this approach is restricted by the characteristics of the hosting language and of its programming model. Obviously, this also requires the application to be (re-)written using the hosting language. The acquisition of expertise

[13]In the case of FTAG, though, one could design a runtime interpreter that dynamically decides the best values for the parameters of the fault-tolerance provisions being executed, where "best" refers to the current environmental conditions.

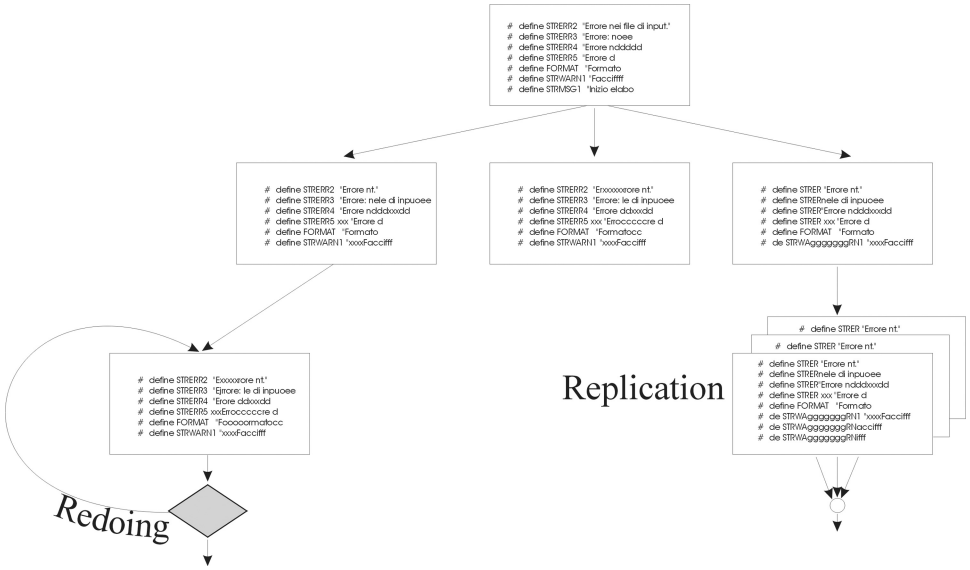

Fig. 10. A fault-tolerant program according to an FTAG system.

in novel design paradigms and languages is likely to have an impact on development costs.

3.4. Aspect-Oriented Programming Languages

Aspect-oriented programming (AOP) [Kiczales et al. 1997] is a programming methodology and structuring technique that explicitly addresses, at a system-wide level, the problem of the best code structure to express different, possibly conflicting design goals such as high performance, optimal memory usage, and dependability.

Indeed, when coding a nonfunctional service within an application (e.g., a system-wide error-handling protocol) using either a procedural or object-oriented programming language, one is required to decompose the original goal in this case a certain degree of dependability into a multiplicity of fragments scattered among a number of procedures or objects. This happens because these programming languages only provide abstraction and composition mechanisms to cleanly support the functional concerns. In other words, specific nonfunctional goals such as high performance cannot be easily captured into a single unit of functionality among those offered by a procedural or object-oriented language, and must be fragmented and intruded into the available units of functionality. As already observed, this code intrusion is detrimental to maintainability and portability of both functional and nonfunctional services (the latter called "aspects" in AOP terminology). These aspects tend to crosscut the system's class and module structure rather than staying well localized within one of these units of functionality, for example, a class. This increases the complexity of the resulting systems.

The main idea of AOP is to use the following tools.

(1) A conventional language (i.e., procedural, object-oriented, or functional programming language) is used to code the basic functionality. The resulting program is called a component program. The program's basic functional units are called components.

(2) A so-called aspect-oriented language is used to implement given aspects by defining specific interconnections ("aspect programs," in the AOP lexicon) among the components in order to address various systemic concerns.
(3) An aspect weaver is used that takes as input both the aspect and component programs and produces with these "weaves" an output program (tangled code) that addresses specific aspects.

The weaver first generates a data flow graph from the component program. In this graph, nodes represent components and edges represent data flowing from one component to another. Next, it executes the aspect programs. These programs edit the graph according to specific goals, collapsing nodes together and adjusting the corresponding code accordingly. Finally, a code generator takes the graph resulting from the previous step as its input and translates it into an actual software package written, for example, for a procedural language such as C. This package is only meant to be compiled and produces the ultimate executable code fulfilling a specific aspect such as, for example, higher dependability.

In a sense, AOP systematically automatizes and supports the process to adapt an existing code so that it fulfils specific aspects. AOP may be defined as a software engineering methodology supporting adaptations in such a way that they do not destroy the original design and do not increase complexity. The original idea of AOP is a clever blending and generalization of the ideas that from the basis, for instance, of optimizing compilers, program transformation systems, MOPs, and literate programming [Knuth 1984].

3.4.1. AspectJ. AspectJ is probably the very first example of aspect-oriented language [Kiczales 2000; Lippert and Videira Lopes 2000]. Developed as a Xerox PARC project, AspectJ can be defined as an aspect-oriented extension to the Java programming language. AspectJ provides its users with the concept of "join points," that is, relevant points in a program's dynamic call graph. Join points are those that mark the code regions that can be manipulated by an aspect weaver (see the previous list). In AspectJ, these points can be:

—method executions,
—constructor calls,
—constructor executions,
—field accesses, and
—exception handlers.

Another extension to Java is AspectJ's support of the design-by-contract methodology [Meyer 1997], where contracts [Hoare 1969] define a set of preconditions, postconditions, and invariants that determine how to use and what to expect from a computational entity.

A study has been carried out to assess the capability of AspectJ as an AOP language supporting exception detection and handling [Lippert and Videira Lopes 2000]. It has been shown how AspectJ can be used to develop so-called plug-and-play exception handlers: libraries of exception handlers that can be plugged into many different applications. This translates into better support for managing different configurations at compile time. This addresses one of the aspects of attribute A, defined in Section 2.3.

3.4.2. AspectWerkz. Recently, a new stream of research activity has been devoted to dynamic AOP. An interesting example of this trend is AspectWerkz [Bonér and

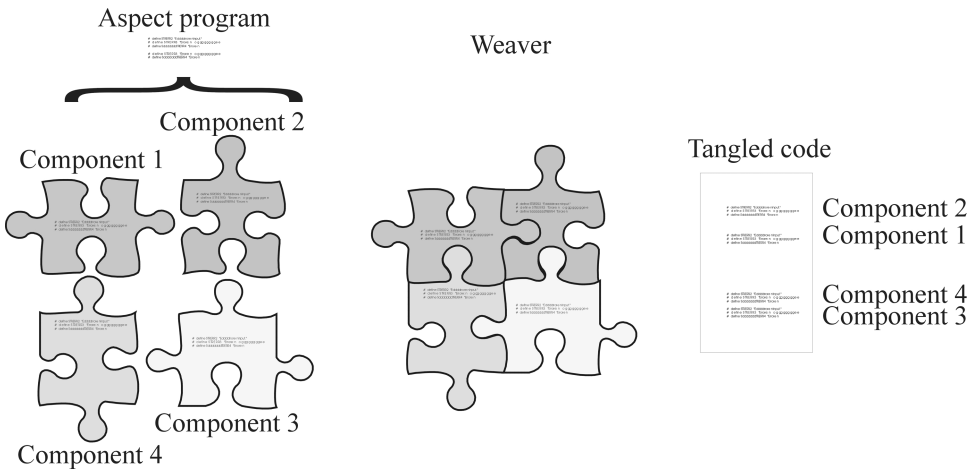

Fig. 11. A fault-tolerant program according to an AOP system.

Vasseur 2004; Vasseur 2004], defined by its authors as "a dynamic, lightweight and high-performant AOP framework for Java" [Bonér 2004]. AspectWerkz utilizes bytecode modification to weave classes at project build-time, class load-time, or runtime. This capability means that the actual semantics of an AspectWerkz code may vary dynamically over time, for example, as a response to environmental changes. This translates into good support towards A.

Recently the members of the AspectJ and AspectWerkz projects have agreed to work together as one team to produce a single aspect-oriented programming platform that builds on their complementary strengths and expertise.

3.4.3. AspectC++. A recent project, AspectC++ [Spinczyk et al. 2005; AspectCpp 2007], proposes an aspect-oriented implementation of C++ which appears to achieve most of the positive properties of the other Java-based approaches and adds to this efficiency and good performance.

3.4.4. Conclusions. Figure 11 synthesizes the main characteristics of AOP: It allows to decompose, select, and assemble components according to different design goals. This has been represented by drawing the components as pieces of a jigsaw puzzle created by the aspect program and assembled by the weaver into the actual source code. AOP addresses explicitly code reengineering, which in principle should allow to reduce considerably the maintenance costs.

AOP is a relatively recent approach to software development. It can in principle address any application domain and can use a procedural, functional, or object-oriented programming language as component language. The isolation and coding of aspects requires extra work and expertise that may be well repaid by the capability of addressing new aspects while keeping a single, unmodified, and general design. This said, we must remark how some researchers have questioned the adequacy of AOP as an effective paradigm for handling failure, at least for the domain of transaction processing [Kienzle and Guerraou 2002].

For the time being it is not yet possible to tell whether AOP will become established as a programming paradigm among academia and industry in the way object-oriented programming has since the '80's. The many qualities of AOP are currently being quantitatively assessed, both with theoretical studies and with practical experience, and the

results seem encouraging. Furthermore, evidence of an increasing interest in AOP is given by the large number of research papers and conferences devoted to this interesting subject.

From a dependability viewpoint, one can observe that AOP exhibits optimal SC ("by construction," in a sense [Kiczales and Mezini 2005]). Furthermore, recent results show that attribute A can in principle reach good values when making use of runtime weaving [Vasseur 2004], often realized by dynamic bytecode manipulation. The work by Ostermann [1999] is an interesting survey on this subject.

Its adequacy at fulfilling attribute SA is indeed debatable also because, to date, no fault tolerance aspect languages have been devised,[14] which may possibly be an interesting research domain.

3.5. The Recovery Metaprogram

The recovery metaprogram (RMP) [Ancona et al. 1990] is a mechanism that alternates the execution of two cooperating processing contexts. The concept behind its architecture can be captured by means of the idea of a debugger or monitor which:

—is scheduled when the application is stopped at some breakpoints;
—executes some program, written in a specific language; and
—finally returns the control to the application context, until the next breakpoint is encountered.

Breakpoints outline portions of code relevant to specific fault tolerance strategies; for instance, breakpoints can be used to specify alternate blocks or acceptance tests of recovery blocks (see Section 3.1.2), while programs are implementations of these strategies (e.g., of recovery blocks or N-version programming). The main benefit of RMP resides in the fact that while breakpoints require a (minimal) intervention of the functional-concerned programmer, RMP scripts can be designed and implemented without intervention, nor even the awareness of the developer. In other words, RMP guarantees a good separation of design concerns. As an example, Figure 12 shows how recovery blocks can be implemented in RMP. We next describe this implementation.

—When the system encounters a breakpoint corresponding to the entrance of a recovery block, the control flows to the RMP, which saves the application program environment and starts the first alternate.
—The execution of the first alternate goes on until its end, marked by another breakpoint. The latter returns the control to RMP, this time in order to execute the acceptance test.
—Should the test succeed, the recovery block is exited; otherwise control goes to the second alternate, and so forth.

Note how the fault tolerance development costs here are basically those for specifying the alternates and acceptance tests, while the remaining complexity is charged to the RMP architecture entirely.

In RMP, the language to express the metaprograms is Hoare's communicating sequential processes (CSP) language [Hoare 1978]. An illustration of an RMP system fault-tolerant program is shown in Figure 13.

[14]For instance, AspectJ only addresses exception error detection and handling. Remarkably enough, the authors of a study on AspectJ and its support to this field conclude [Lippert and Videira Lopes 2000] that "whether the properties of AspectJ [documented in this article] lead to programs with fewer implementation errors and that can be changed easier, is still an open research topic that will require serious usability studies as AOP matures."

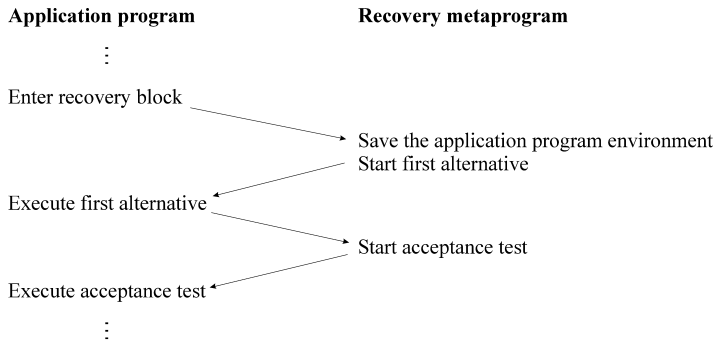

Fig. 12. Control flow between the application program and RMP while executing a fault tolerance strategy based on recovery blocks.

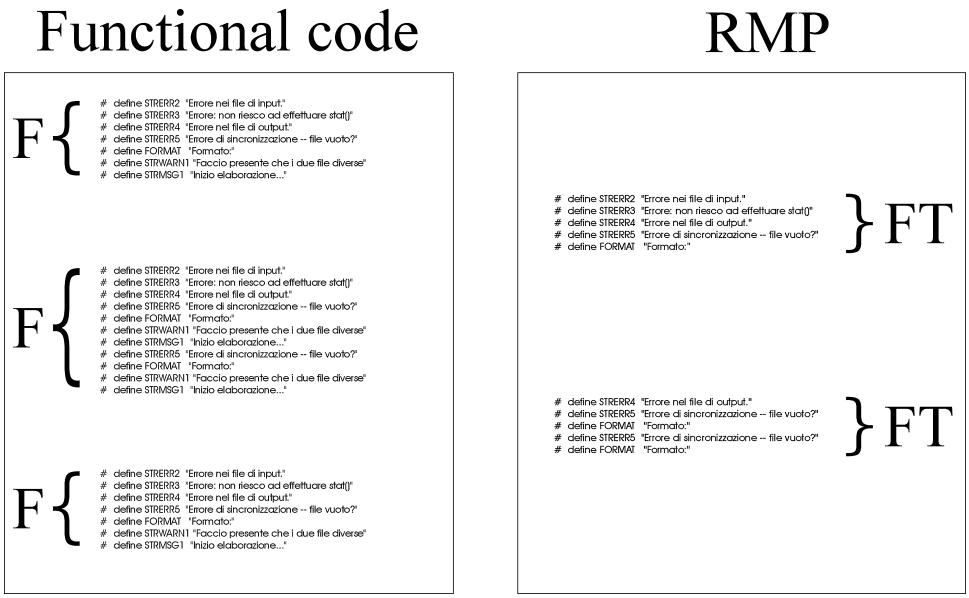

Fig. 13. A fault-tolerant program according to the RMP system.

3.6. Conclusions

Figure 9 synthesizes the main characteristics of RMP: The fault tolerance code is in this case both logically and physically distinct from the functional code, which means that the coding complexity and costs are considerably reduced.

In the RMP approach, all technicalities related to management of the fault tolerance provisions are coded in a separate programming context. Even the language to code the provisions may be different from the one used to express the functional aspects of the application. One can conclude that RMP is characterized by optimal SC.

The design choice of using CSP to code metaprograms influences attribute SA negatively. Choosing a preexistent formalism clearly presents many practical advantages, though it means adopting a fixed, immutable syntactical structure to express the fault-tolerance strategies. The choice of a preexisting general-purpose distributed programming language such as CSP is therefore questionable, as it appears to be rather difficult

Table I. A Summary of the Qualitative Assessments Proposed in Section 3

Section	Approach	SC	SA	A
3.1.1	SV	poor	very limited	poor
3.1.2	MV (recovery blocks)	poor	poor	poor
3.1.2	MV (NVP)	good	poor	poor
3.2	MOP	optimal	positive?	positive
3.3.1	EL	positive	poor	poor
3.3.2	DL	positive	optimal	poor
3.4	AOP	optimal	positive?	good
3.5	RMP	optimal	very limited	positive

MV has been differentiated into recovery blocks (RB) and NVP. Also, EL is the approach of Section 3.3.1, while DL is that of Section 3.3.2.

or at least cumbersome to use it to express at least some of the fault tolerance provisions. For instance, RMP proves an effective linguistic structure to express strategies such as recovery blocks and N-version programming, where the main components are coarse-grain processes to be arranged into complex fault tolerance structures. Because of the choice of a preexisting language such as CSP, RMP appears not to be the best choice for representing provisions such as, for example, atomic actions [Jalote and Campbell 1985]. This translates into very limited SA.

Our conjecture is that the coexistence of two separate layers for the functional and nonfunctional aspects could have been better exploited to reach the best of the two approaches: that of using a widespread programming language (e.g., C), for expressing the functional aspect, while devising a custom language for dealing with nonfunctional requirements (e.g., a language specially designed to express error recovery strategies).

Satisfactory values for attribute A cannot be reached with the only RMP system developed so far, because it does not foresee any dynamic management of the executable code. Nevertheless, it is definitely possible to design systems in which, for example, the recovery metaprogram changes dynamically so as to compensate for changes in the environment, or other changes. This is the strategy used in De Florio and Blondia [2005] to set-up a system structure for adaptive mobile applications. Hence, we chose to evaluate as positive the value of A for the RMP *approach*.

RMP appears characterized by a large overhead due to frequent context switching between the main application and the recovery metaprogram [Randell and Xu 1995]. Runtime requirements may be jeopardized by these large overheads, especially when it is difficult to establish time bounds for their extent. No other restrictions appear to be imposed by RMP on the target application domain.

4. CONCLUSIONS

Five classes of system structure for ALFT have been described and critically reviewed, qualitatively, with respect to the structural attributes SC, SA, and A. Table I summarizes the results of this survey, providing a comparison of the various approaches. As can be seen from these summaries, no single approach exists today that provide an optimal solution to the problems cumulatively referred to as the system structure for application-level fault tolerance. We are currently working towards the definition of new models for application-level fault tolerance reaching high values of the three attributes. A prototype system is described in De Florio and Blondia [2007b].

This article has also highlighted the positive and negative aspects of an evolutionary solution (using a preexisting language) with respect to a "revolutionary" approach based on devising a custom-made, ad hoc language. As a consequence of these observations, it has been conjectured that using an approach based on two languages (one covering functional concerns and the other the fault-tolerance ones), it may be possible

to address, within one efficacious linguistic structure, the widest set of fault tolerance provisions, thus providing optimal values for the three structural attributes. This conjecture is currently under verification [De Florio and Blondia 2005] in the framework of European project ARFLEX (Adaptive Robots for Flexible Manufacturing Systems) and IBBT project QoE (end-to-end Quality of Experience).

ACKNOWLEDGMENT

We would like to express our gratitude to the Editor for the many insightful remarks and suggestions.

REFERENCES

AKSIT, M., DIJKSTRA, J., AND TRIPATHI, A. 1991. Atomic delegation: Object-Oriented transactions. *IEEE Softw. 8*, 2, 84–92.

AMMANN, P. E. AND KNIGHT, J. C. 1988. Data diversity: An approach to software fault tolerance. *IEEE Trans. Comput. 37*, 4, 418–425.

ANCONA, M., DODERO, G., GIANNUZZI, V., CLEMATIS, A., AND FERNANDEZ, E. B. 1990. A system architecture for fault tolerance in concurrent software. *IEEE Comput. 23*, 10 (Oct.), 23–32.

ANDERSON, B. G. AND SHASHA, D. 1991. Persistent Linda: Linda + transactions + query processing. In *Proceedings of the Workshop on Research Directions in High-Level Parallel Programming Languages*, J. Banâtre and D. Le Métayer, eds. Lecture Notes in Computer Science, vol. 54. Springer, 93–109.

ANDERSON, T., BARRETT, P., HALLIWELL, D., AND MOULDING, M. 1985. Software fault tolerance: An evaluation. *IEEE Trans. Softw. Eng. 11*, 2, 1502–1510.

ANDREWS, G. R. AND OLSSON, L. L. 1993. *The SR Programming Language: Concurrency in Practice*. Benjamin-Cummings.

ASPECTCPP. 2007. The home of AspectC++. www.aspectc.org (last accessed Aug. 21, 2007).

AVIŽIENIS, A. 1995. The methodology of *N*-version programming. In *Software Fault Tolerance*, M. Lyu, ed. John Wiley and Sons, New York, Chapter 2, 23–46.

AVIŽIENIS, A. 1985. The *N*-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng. 11*, 1491–1501.

AVIŽIENIS, A., LAPRIE, J.-C., AND RANDELL, B. 2004a. Dependability and its threats: A taxonomy. In *Proceedings of the IFIP 18th World Computer Congress*. Kluwer Academic, 91–120.

AVIŽIENIS, A., LAPRIE, J.-C., RANDELL, B., AND LANDWEHR, C. 2004b. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Depend. Secure Comput. 1*, 1, 11–33.

BAKKEN, D. E. AND SCHLICHTING, R. D. 1995. Supporting fault-tolerant parallel programming in Linda. *IEEE Trans. Parallel Distrib. Syst. 6*, 3 (Mar.), 287–302.

BAO, Y., SUN, X., AND TRIVEDI, K. 2003. Adaptive software rejuvenation: Degradation models and rejuvenation schemes. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society.

BERNDT, D. 1989. *C-Linda Reference Manual*. Scientific Computing Associates.

BIRMAN, K., SCHIPER, A., AND STEPHENSON, P. 1991. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst. 9*, 3 (Aug.).

BIRRELL, A. D. AND NELSON, B. J. 1984. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 39–59.

BONDAVALLI, A., CHIARADONNA, S., DI GIANDOMENICO, F., AND GRANDONI, F. 1997. Discriminating fault rate and persistency to improve fault treatment. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing Systems (FTCS)*, Munich, Germany. IEEE Computer Society Press, 354–362.

BONÉR, J. 2004. AspectWerkz—Dynamic AOP for Java. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*.

BONÉR, J. AND VASSEUR, A. 2004. Dynamic AOP: SOA for the application. Tutorial presented at *the Annual BEA eWorld Technology Conference*.

CANNON, S. AND DUNN, D. 1992. A high-level model for the development of fault-tolerant parallel and distributed systems. Tech. Rep. A0192, Department of Computer Science, Utah State University. August.

CARRIERO, N. AND GELERNTER, D. 1989a. How to write parallel programs: A guide to the perplexed. *ACM Comput. Surv. 21*, 323–357.

CARRIERO, N. AND GELERNTER, D. 1989b. Linda in context. *Commun. ACM 32*, 4, 444–458.

CRISTIAN, F. 1995. Exception handling. In *Software Fault Tolerance*, M. Lyu, ed. Wiley, 81–107.

DE FLORIO, V. 1998. The DIR net: A distributed system for detection, isolation, and recovery. Tech. Rep. ESAT/ACCA/1998/1, University of Leuven. October.

DE FLORIO, V. 1997a. The EFTOS recovery language. Tech. Rep. ESAT/ACCA/1997/4, University of Leuven. December.

DE FLORIO, V. 1997b. The voting farm—A distributed class for software voting. Tech. Rep. ESAT/ACCA/1997/3, University of Leuven. June.

DE FLORIO, V. AND BLONDIA, C. 2007a. Adaptation as a new requirement for software engineering. In *Proceedings of the 1st IEEE Workshop on Adaptive and Dependable Mission- and Business-Critical Mobile Systems (ADAMUS) at the IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMom)*, Helsinki, Finland.

DE FLORIO, V. AND BLONDIA, C. 2007b. Reflective and refractive variables: A model for effective and maintainable adaptive-and-dependable software. In *Proceedings of the 33rd Euromicro Conference on Software Engineering and Advanced Applications (SEEA), Software Process and Product Improvement Track (SPPI)*, Lübeck, Germany. IEEE Computer Society Press.

DE FLORIO, V. AND BLONDIA, C. 2005. A system structure for adaptive mobile applications. In *Proceedings of the 6th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. Taormina - Giardini Naxos, Italy, 270–275.

DE FLORIO, V. AND DECONINCK, G. 2002. $\mathcal{R}\mathcal{E}\mathcal{L}$: A fault tolerance linguistic structure for distributed applications. In *Proceedings of the 9th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, Lund, Sweden. IEEE Computer Society Press.

DE FLORIO, V., DECONINCK, G., LAUWEREINS, R., AND GRAEBER, S. 2001. Design and implementation of a data stabilizing software tool. In *Proceedings of the 9th Euromicro Workshop on Parallel and Distributed Processing (Euro-PDP)*, Mantova, Italy. IEEE Computer Society Press.

DE FLORIO, V., DECONINCK, G., AND LAUWEREINS, R. 2000. An algorithm for tolerating crash failures in distributed systems. In *Proceedings of the 7th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, Edinburgh, Scotland. IEEE Computer Society Press, 9–17.

DE FLORIO, V., DECONINCK, G., AND LAUWEREINS, R. 1999. An application-level dependable technique for farmer-worker parallel programs. *Informatica 23*, 2 (May), 275–281.

DE FLORIO, V., DECONINCK, G., AND LAUWEREINS, R. 1998a. The EFTOS voting farm: A software tool for fault masking in message passing parallel environments. In *Proceedings of the 24th Euromicro Conference (Euromicro), Workshop on Dependable Computing Systems*, Västerås, Sweden. IEEE Computer Society Press, 379–386.

DE FLORIO, V., DECONINCK, G., AND LAUWEREINS, R. 1998b. Software tool combining fault masking with user-defined recovery strategies. *IEE Proc. Softw. 145*, 6 (Dec.), 203–211. *SI Depend. Comput. Syst.* IEE in association with the British Computer Society.

DE FLORIO, V., DECONINCK, G., TRUYENS, M., ROSSEEL, W., AND LAUWEREINS, R. 1998c. A hypermedia distributed application for monitoring and fault-injection in embedded fault-tolerant parallel programs. In *Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing (Euro-PDP)*, Madrid, Spain. IEEE Computer Society Press, 349–355.

DE FLORIO, V., MURGOLO, F. P., AND SPINELLI, V. 1994. PvmLinda: Integration of two different computation paradigms. In *Proceedings of the 1st Euromicro Conference on Massively Parallel Computing Systems (MPCS)*, Ischia, Italy. IEEE Computer Society Press, 488–496.

DENNING, P. J. 1976. Fault tolerant operating systems. *ACM Comput. Surv. 8*, 4, 359–389.

EBNENASIR, A. AND KULKARNI, S. 2004. Hierarchical presynthesized components for automatic addition of fault tolerance: A case study. In *Proceedings of the Specification and Verification of Component-Based Systems Workshop (SAVCBS) at ACM SIGSOFT/FSE-12*.

ECKHARDT, D. E. AND LEE, L. D. 1985. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Trans. Softw. Eng. 11*, 12 (Dec.), 1511–1517.

ECKHARDT, D. E., CAGLAYAN, A. K., KNIGHT, J. C., LEE, L. D., MCALLISTER, D. F., VOUK, M. A., AND KELLY, J. P. J. 1991. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE Trans. Softw. Eng. 17*, 7 (Jul.), 692–702.

EFTHIVOULIDIS, G., VERENTZIOTIS, E. A., MELIONES, A. N., VARVARIGOU, T. A., KONTIZAS, A., DECONINCK, G., AND DE FLORIO, V. 1998. Fault tolerant communication in embedded supercomputing. *IEEE Micro SI Fault Tolerance 18*, 5 (Sept.-Oct.), 42–52.

ELNOZAHY, E. N., ALVISI, L., WANG, Y.-M., AND JOHNSON, D. B. J. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv. 34*, 3, 375–408.

FABRE, J.-C. AND PÉRENNOU, T. 1998. A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach. *IEEE Trans. Comput. 47*, 1 (Jan.), 78–95.

FABRE, J.-C. AND PÉRENNOU, T. 1996. FRIENDS: A flexible architecture for implementing fault tolerant and secure applications. In *Proceedings of the 2nd European Dependable Computing Conference (EDCC)*. Taormina, Italy.

GELERNTER, D. 1985. Generative communication in Linda. *ACM Trans. Program. Lang. Syst. 7*, 1 (Jan.).

GLANDRUP, M. H. J. 1995. Extending C++ using the concepts of composition filters. M.S. thesis, Department of Computer Science, University of Twente, Enschede, The Netherlands.

GREEN, P. A. 1997. The art of creating reliable software-based systems using off-the-shelf software components. In *Proceedings of the 16th Symposium on Reliable Distributed Systems (SRDS)*. Durham, NC.

GUERRAOUI, R. AND SCHIPER, A. 1997. Software-Based replication for fault tolerance. *IEEE Comput. 30*, 4 (Apr.), 68–74.

HOARE, C. A. R. 1978. Communicating sequential processes. *Commun. ACM 21*, 667–677.

HOARE, C. A. R. 1969. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct.), 576–580.

HORNING, J. J. 1998. ACM fellow profile—James Jay (Jim) Horning. *ACM Softw. Eng. Not. 23*, 4 (Jul.).

HUANG, Y. AND KINTALA, C. M. 1995. Software fault tolerance in the application layer. In *Software Fault Tolerance*, M. Lyu, ed. John Wiley and Sons, New York, Chapter 10, 231–248.

HUANG, Y., KINTALA, C. M., BERNSTEIN, L., AND WANG, Y. 1996. Components for software fault tolerance and rejuvenation. *AT&T Tech. J.*, 29–37.

HUANG, Y., KINTALA, C., KOLETTIS, N., AND FULTON, N. D. 1995. Software rejuvenation: Analysis, module and applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS)*.

INQUIRY BOARD REPORT. 1996. ARIANE 5—Flight 501 failure. http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html.

JALOTE, P. AND CAMPBELL, R. H. 1985. Atomic actions in concurrent systems. In *Proceedings of the 5th International Conference on Distributed Computing Systems (ICDCS)*, Denver, CO. ISBN 0-8186-0617-7.

JOHNSON, B. W. 1989. *Design and Analysis of Fault-Tolerant Digital Systems*. Addison-Wesley, New York.

KAMBHATLA, S. 1991. Replication issues for a distributed and highly available Linda tuple space. M.S. thesis, Department of Computer Science, Oregon Graduate Institute.

KICZALES, G. 2000. AspectJ™: Aspect-Oriented programming using Java™ technology. In *Proceedings of the Sun's Worldwide Java Developer Conference (JavaOne)*. San Francisco, CA. Slides available at URL http://aspectj.org/servlets/AJSite?channel=documentation&subChannel=papersAndSlides.

KICZALES, G. AND MEZINI, M. 2005. Separation of concerns with procedures, annotations, advice and pointcuts. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, Springer.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., VIDEIRA LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-Oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* Lecture Notes in Computer Science, vol. 1241. Springer, Berlin, Finland.

KICZALES, G., DES RIVIÈRES, J., AND BOBROW, D. G. 1991. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, MA.

KIENZLE, J. AND GUERRAOU, R. 2002. AOP: Does it make sense? The case of concurrency and failures. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)* 37–61.

KIM, E. 1996. *CGI Developer's Guide*. SAMS.NET.

KIM, K. AND WELCH, H. 1989. Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications. *IEEE Trans. Comput. 38*, 5, 626–636.

KLEINROCK, L. 1975. *Queueing Systems* (2 volumes). John Wiley and Sons.

KNUTH, D. E. 1984. Literate programming. *The Comp. J. 27*, 97–111.

KULKARNI, S. S. AND ARORA, A. 2000. Automating the addition of fault tolerance. Tech. Rep. MSU-CSE-00-13, Department of Computer Science, Michigan State University, East Lansing, Michigan. June.

LAPRIE, J.-C. 1998. Dependability of computer systems: From concepts to limits. In *Proceedings of the IFIP International Workshop on Dependable Computing and Its Applications (DCIA)*. Johannesburg, South Africa.

LAPRIE, J.-C. 1995. Dependability—Its attributes, impairments and means. In *Predictably Dependable Computing Systems*, B. Randell et al., eds. ESPRIT Basic Research Series. Springer, Berlin, 3–18.

LAPRIE, J.-C. 1992. *Dependability: Basic Concepts and Terminology in English, French, German, Italian and Japanese*. Dependable Computing and Fault Tolerance, vol. 5. Springer Verlag, Wien.

LAMPORT, L., SHOSTAK, R., AND PEASE, M. 1982. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4*, 3 (Jul.), 384–401.

LE LANN, G. 1996. The Ariane 5 flight 501 failure—A case study in system engineering for computing systems. Tech. Rep. 3079, INRIA. December.

LEVESON, N. G. 1995. *Safeware: Systems Safety and Computers*. Addison-Wesley.

LIPPERT, M. AND VIDEIRA LOPES, C. 2000. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*. Limmerick, Ireland.

LISKOV, B. 1988. Distributed programming in Argus. *Commun. ACM 31*, 3 (Mar.), 300–312.

LORCZAK, P. R., CAGLAYAN, A. K., AND ECKHARDT, D. E. 1989. A theoretical investigation of generalized voters for redundant systems. In *Proceedings of the 19th International Symposium on Fault-Tolerant Computing (FTCS)*, Chicago, IL, 444–451.

LYU, M. R. 1998a. Design, testing, and evaluation techniques for software reliability engineering. In *Proceedings of the 24th Euromicro Conference on Engineering Systems and Software for the Next Decade (Euromicro), Workshop on Dependable Computing Systems* (Keynote speech), Västerås, Sweden. IEEE Computer Society Press, xxxix–xlvi.

LYU, M. R. 1998b. Reliability-Oriented software engineering: Design, testing and evaluation techniques. *IEE Proc. Softw. SI Depend. Comput. Syst. 145*, 6 (Dec.), 191–197.

LYU, M., ED. 1996. *Handbook of Software Reliability Engineering*. IEEE Computer Society Press and McGraw-Hill.

LYU, M. 1995. *Software Fault Tolerance*. John Wiley and Sons, New York.

MAES, P. 1987. Concepts and experiments in computational reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Orlando, FL, 147–155.

MASUHARA, H., MATSUOKA, S., WATANABE, T., AND YONEZAWA, A. 1992. Object-Oriented concurrent reflective languages can be implemented efficiently. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 127–144.

MEYER, B. 1997. *Fault-Tolerant Computer Systems Design*. Prentice-Hall, NJ.

NRC. 1993. Switch focus team report. Tech. Rep., National Reliability Council. June.

OMG. 1998. Fault tolerant CORBA using entity redundancy. Tech. Rep. Request for Proposal, Object Management Group (OMG). December.

OSTERMANN, K. 1999. Towards a composition taxonomy. Tech. Rep., Siemens AG CT SE 2.

PAAKKI, J. 1995. Attribute grammar paradigms: A high-level methodology in language implementation. *ACM Comput. Surv. 27*, 2 (Jun.).

PARRINGTON, G. D. 1990. Reliable distributed programming in C++: The Arjuna approach. In *Proceedings of the 2nd Usenix C++ Conference*, San Francisco, 37–50.

PARSYTEC. 1996. *Parsytec CC Series—Cognitive Computing*. Parsytec GmbH, Aachen, Germany.

PATTERSON, L. I., TURNER, R. S., HYATT, R. M., AND REILLY, K. D. 1993. Construction of a fault-tolerant distributed tuple-space. In *Proceedings of the ACM SIGAPP Symposium on Applied Computing*.

POWELL, D. 1997. Preliminary definition of the GUARDS architecture. Tech. Rep. 96277, LAAS-CNRS. January.

POWELL, D., ARLAT, J., BEUS-DUKIC, L., BONDAVALLI, A., COPPOLA, P., FANTECHI, A., JENN, E., RABÉJAC, C., AND WELLINGS, A. 1999. GUARDS: A generic upgradable architecture for real-time dependable systems. *IEEE Trans. Parallel Distrib. Syst. 10*, 6 (Jun.), 580–599.

PRADHAN, D. K. 1996. *Fault-Tolerant Computer Systems Design*. Prentice-Hall, Upper Saddle River, NJ.

RANDELL, B. 1975. System structure for software fault tolerance. *IEEE Trans. Softw. Eng. 1*, 220–232.

RANDELL, B. AND XU, J. 1995. The evolution of the recovery block concept. In *Software Fault Tolerance*, M. Lyu, ed. John Wiley and Sons, New York, Chapter 1, 1–21.

ROBBEN, B. 1999. Language technology and metalevel architectures for distributed objects. Ph.D. thesis, Department of Computer Science, University of Leuven.

SALTZER, J. H., REED, D. P., AND CLARK, D. D. 1984. End-to-End arguments in system design. *ACM Trans. Comput. Syst. 2*, 4, 277–288.

SCHLICHTING, R. D. AND THOMAS, V. T. 1995. Programming language support for writing fault-tolerant distributed software. *IEEE Trans. Comput. SI Fault-Toler. Comput. 44*, 2 (Feb.), 203–212.

SCHNEIDER, F. 1990. Implementing fault-tolerant services using the state machine approach. *ACM Comput. Surv. 22*, 4, 299–319.

SCHOINAS, G. 1991. Issues on the implementation of POSYBL: A free Linda implementation for Unix networks. Tech. Rep., Department of Computer Science, University of Crete.

SCOTT, R., GAULT, J., AND MCALLISTER, D. 1985. The consensus recovery block. In *Proceedings of the Total System Reliability Symposium*, 74–85.

SHANNON, C. E., WINER, A. D., AND SLOANE, N. J. A. 1993. *Claude Elwood Shannon: Collected Papers*. Amazon.

SHRIVASTAVA, S. 1995. Lessons learned from building and using the Arjuna distributed programming system. In *Theory and Practice in Distributed Systems*. Lecture Notes in Computer Science, vol. 938. Springer.

SHRIVASTAVA, S. 1978. Sequential Pascal with recovery blocks. *Softw. Pract. Exper. 8*, 177–185.

SIBLEY, E. H. 1998. Computer security, fault tolerance, and software assurance: From needs to solutions. In *Proceedings of the Workshops on Computer Security, Fault Tolerance, and Software Assurance: From Needs to Solutions*, York, UK.

SIEWIOREK, D. P. AND SWARZ, R. S. 1992. *Reliable Computer Systems Design and Implementation*. Digital Press.

SPINCZYK, O., LOHMANN, D., AND URBAN, M. 2005. AspectC++: An AOP extension for C++. *Softw. Dev. J.*, 68–76.

SUZUKI, M., KATAYAMA, T., AND SCHLICHTING, R. D. 1996. FTAG: A functional and attribute based model for writing fault-tolerant software. Tech. Rep. TR 96-6, Department of Computer Science, The University of Arizona. May.

TANENBAUM, A. S. 1990. *Structured Computer Organization*, 3rd ed. Prentice-Hall.

TAYLOR, D. J., MORGAN, D. E., AND BLACK, J. P. 1980. Redundancy in data structures: Improving software fault tolerance. *IEEE Trans. Softw. Eng. 6*, 6 (Nov.), 585–594.

VAN ACHTEREN, T. 1997. Object georienteerde afleiding van metaobjecten voor foutbestendigheid in de friends omgeving. M.S. thesis, Department of Electrical Engineering, University of Leuven. In Flemish.

VASSEUR, A. 2004. Dynamic AOP and runtime weaving for Java—How does AspectWerkz address it? In *Proceedings of AOSD 2004, Dynamic AOP WorkShop*.

WEIK, M. H. 1961. The ENIAC story. *ORDNANCE J. Amer. Ordnance Assoc*. http://ftp.arl.mil/~mike/comphist/eniac-story.html.

WIENER, L. 1993. *Digital Woes. Why We Should Not Depend on Software*. Addison-Wesley.

XU, A. AND LISKOV, B. 1989. A design for a fault-tolerant, distributed implementation of Linda. In *Proceedings of the 19th International IEEE Symposium on Fault-Tolerant Computing (FTCS)*. IEEE, 199–206.

ZAWINSKI, J. 1994. Remote control of UNIX Netscape. Tech. Rep., Netscape Communications Corp. December. wp.netscape.com/newsref/std/x-remote.html.

Received September 2003; revised July 2005; accepted August 2007