A Survey on Linguistic Structures for Application-level Fault-Tolerance

VINCENZO DE FLORIO and CHRIS BLONDIA
University of Antwerp
Department of Mathematics and Computer Science
Performance Analysis of Telecommunication Systems group
Middelheimlaan 1, 2020 Antwerp, Belgium, *and*
Interdisciplinary institute for BroadBand Technology
Gaston Crommenlaan 8, 9050 Ghent-Ledeberg, Belgium

The structures for the expression of fault-tolerance provisions into the application software are the central topic of this paper. Structuring techniques answer the questions "How to incorporate fault-tolerance in the application layer of a computer program" and "How to manage the fault-tolerance code". As such, they provide 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 above mentioned 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.2.7 [**Software Engineering**]: Software Architectures—*Languages*; *Domain-specific architectures*; D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*; C.4 [**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.

1. INTRODUCTION

Research in fault-tolerance focussed for decades on *hardware* fault-tolerance, i.e., 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 one effective solution to reach the required availability and data integrity demanded by 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 way to go: 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, e.g., for the hardware [Pradhan 1996], the operating system [Denning 1976], or the 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 an open one. To the best of our knowledge, no detailed critical survey of the available solutions exists. Through this paper 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], like e.g. in CORBA-FT [(OMG) 1998]. Likewise this text does not include approaches like the one in [Ebnenasir and Kulkarni 2004], where the focus is on automating the transformation of a given fault-intolerant program into a fault-tolerant program. 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 the awareness of a need for an optimal solution to it: Indeed, the current lack of a simple and coherent system structure for software fault-tolerance engineering able to provide the designer with effective support towards fulfilling goals such as maintainability, re-usability, and service portability of fault-tolerant software manifests itself as a true bottleneck for system development.

The structure of this paper is as follows: in Sect. 2 we explain the reasons behind the need for system-level fault-tolerance. There 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 of which is 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 this section. Section 4 finally summarizes our conclusions and provides a comparison of the reviewed approaches.

2. RATIONALE

No artifact in human history has ever permeated so many aspects of human life as the computer did during the last half a century. An outstanding aspect of this success is certainly given by the overwhelming increase in computer performance, though probably the most important one is the growth in complexity and crucial character of the roles nowadays assigned to computers: Human society more and more expects and relies on good quality of complex services supplied by computers. More and more these services become vital, in the sense that ever more often a lack of timely delivery can impact severely on capitals, the environment, and even human lives.

In other words, if in the early days of modern computing it was to some extent acceptable that outages and wrong results occurred rather often[1], being the main

[1]This excerpt from a report on the ENIAC activity [Weik 1961] gives an idea of how dependable computers were in 1947: "power 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

role of computers basically that of a fast solver of numerical problems, today the criticality associated with many tasks dependent on computers does require strong guarantees for properties such as availability and data integrity.

Consequence of this growth in complexity and criticality is the need for

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

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

This paper focusses in particular on **fault-tolerance**, that is, how to ensure a service up to fulfilling the system's function even in the presence of "faults" [Laprie 1998], the latter being events that have their origin inside or outside the system boundaries and possibly introduce unexpected deviations of the system state.

2.1 A Need for Software Fault-Tolerance

Research in fault-tolerance concentrated for many years on *hardware* fault-tolerance, i.e., 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 requirements of availability and data integrity demanded by modern complex computer services. Probably the first researcher who realized that this was far from being true was B. Randell who in [Randell 1975] questioned hardware fault-tolerance as the only approach to pursue—in the cited paper he states:

> "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 in the hardware layer [Lyu 1998a; 1998b; Huang and Kintala 1995; Wiener 1993; Randell 1975]. This state of things could only be reached by exploiting a powerful conceptual tool for managing complexity in a flexible and effective way, i.e., devising hierarchies of sophisticated abstract machines [Tanenbaum 1990]. This translates in implementing software with high-level computer languages lying on top of other software strata—the device drivers layers, the basic services kernel, the operating system, the run-time support of the involved programming languages, and so forth.

Partitioning the complexity into stacks of software layers allowed the implementor to focus exclusively on the high-level aspects of their problems, and hence it allowed to manage a larger and larger degree of complexity. But *though made transparent, still this complexity is 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 no involved abstract machine nor the actual hardware be affected by faults.

time remained at about 100 hours a week during the last 6 years of the ENIAC's use", i.e., an availability of about 60%!

Unfortunately, as in real life faults do occur, the corresponding deviations are likely to jeopardise the system's function, also propagating from one layer to the other, unless appropriate means are taken to avoid in the first place, or to remove, or to tolerate these faults. In particular, faults may also occur in the **application layer**, that is, in the abstract machine on top of the software hierarchy[2]. These faults, possibly having their origin at design time, or during operation, or while interacting with the environment, *are not different in the extent of their consequences from those faults originating, e.g., in the hardware or the operating system.*

An efficacious argument to bring evidence to the above statement is the case of the so-called "millennium bug", i.e., the most popular class of design faults that ever showed up in the history of computing technologies, also known as "the year 2000 problem" or just "Y2K": most of the software still in use today was developed using a standard where dates are coded in a 6-digit format. According to this standard, two digits were considered as enough to represent the year. Unfortunately this translates into the impossibility to distinguish, e.g., year 2000 from year 1900. Near year 2000 this was finally recognised as the possible single cause for an unpredictably large number of failures when calculating time elapsed between two calendar dates. The adoption of the above mentioned standard for representing dates resulted in a hidden, forgotten design fault, never considered nor tested by application programmers. As society got closer and closer to year 2000, the possible presence of this design fault became a threat that seemed to jeopardise many crucial functions of our society—those functions that had been appointed to programs using calendar dates, e.g. in utilities, transportation, health care, communication, or public administration. Luckily the expected many and crucial system failures due to this single application-level fault [Hansen et al. 1999] were not so many and not that crucial, though probably for the first time the whole society became aware of the extent of the relevance of dependability in software.

These facts and the above reasoning suggest that, the higher the level of abstraction, the higher the complexity of the algorithms into play and the consequent error proneness of the involved (real or abstract) machines. As a conclusion, 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 levels, including the application layer.*

2.2 Software Fault-Tolerance in the Application Layer

The need of software fault-tolerance provisions, located in the application layer, is supported by studies that showed 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; Laprie 1998]; 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, e.g., client-server applications, are often characterised by a loosely coupled architecture whose global structure is in

[2]In what follows, the application layer is to be 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 is running.

general more prone to failures[3]. Due to the complex and temporal nature of 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 of 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 object orientation. Many object-oriented applications are indeed built from *reusable components* the sources of which are unknown to the application developers. The object abstraction 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 though translates the application in a sort of collection of reused, pre-existing objects made by third parties. The reliability of these components and therefore their impact on the overall reliability of the user application is often unknown, up to the point 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[4] [Inquiry Board Report 1996].

But 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, Reed and Clark [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 (for instance, the communication network).

This does not mean that everything should be done at the application level—fault-tolerance strategies in the underlying hardware and operating system can

[3]As Leslie Lamport efficaciously synthesised in his quotation, "a distributed system is one in which I cannot get something done because a machine I've never heard of is down".

[4]Within the Ariane 5 programme, it was decided to reuse the long-tested software used in the Ariane 4 programme. Such software had been thoroughly tested and was compliant to Ariane 4 specifications. Unfortunately, specifications for Ariane 5 were different—in particular, they stated that the launcher had a trajectory characterised by considerably higher horizontal velocity values. A dormant design fault, related to an overflow while casting a 64-bit floating point value representing horizontal velocity to a 16-bit signed integer was never unravelled simply because, given the moderate horizontal velocity values of Ariane 4, such an overflow would have never occurred. Reusing this same software, including the routine affected by this dormant design fault, in both the primary and the active backup Inertial Reference Systems that were in use in the Ariane 5 flight 501 the morning of 4 June 1996, triggered almost at the same time the failure of two Inertial Reference Systems. Diagnostic information produced by that component was then interpreted as correct flight information, making the launcher veer off its flight path, then initiate a self-destruction procedure, and eventually explode. It is worth mentioning that the faulty routine served an Ariane 4 specific functionality, that was of no use in Ariane 5. It had been nevertheless included in order to reuse exactly the same software adopted in the previous programme. Such a software was indeed considered design fault free. This failure entailed a loss of about 1.9 billion French francs (approximately 0.37 billion Euros) and a delay of about one year for the Ariane 5 programme [Le Lann 1996].

have a strong impact on a system's performance. However, an extraordinarily reliable communication system, that guarantees that no packet is lost, duplicated, or corrupted, nor delivered to the wrong addressee, does not reduce the burden of the application program to ensure 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 characterised 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].

2.3 Strategies, Problems, and Key Properties

The above 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 Chapter 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.
(3) How to manage the fault-tolerance code.

Problem 1 is also known as the problem of the **system structure to software fault-tolerance**, first proposed by B. Randell in [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 that addresses the functional requirements and the application code that addresses the fault-tolerance requirements are mixed up into one large and complex application software.

—This 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 the**

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 efficaciously captures this concept:

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

Indeed, as explained in Sect. 2.3.1, a non-optimal answer to Problem 2 may

—require a high degree of redundancy, and
—rapidly consume large amounts of the available redundancy,

which at the same time would increase the costs and reduce 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-tolerance code is managed*, at compile time as well as at run time. Evidence to this statement can be brought by observing the following facts:

—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 is to run[5]. In other words, depending on the target environments, 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 above mentioned choices, namely the parameters of the adopted provisions. Effective support towards the management of the parametrisation of the fault-tolerance code and, in general, of its maintenance, could guarantee **fault-tolerance software reuse**.
—The *dynamic* (run-time) adaptation of the fault-tolerance code and of its parameters would allow the application to also stand unexpected changes in the environment, such as an increase of temperature triggered by a failure of the cooling system. These events may trigger unexpected faults. Furthermore, the ever increasing diffusion of *mobile software components*, coupled with the increasing need for dependability, will require more and more the capability to guarantee an agreed quality of service even when the environment changes during the run-time because of mobility.

Therefore, off-line and on-line (dynamic) management of fault-tolerance provisions and their parameters may be an important requirement for any satisfactory solution

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

of Problem 3. As further motivated in Sect. 2.3.1, ideally, the fault-tolerance code should *adapt* itself to the current environment. Furthermore, any satisfactory management approach should not overly increase the complexity of the application—which would be detrimental to dependability. Let us call this property **Adaptability** (A).

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

The various approaches to ALFT surveyed in Section 3 provide different system structures to solve the above mentioned problems. The three structural attributes are used in that section in order to provide a qualitative assessment with respect to various application requirements. The structural attributes constitute, in a sense, a *base* with which to perform this assessment. One of the major conclusions of that survey is that *none* of the surveyed approaches is capable to provide 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, while only partial solutions exist, for instance, when dealing with the class of distributed or parallel applications *not based on the object model.*

The above matter of facts has 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 above 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 channel by increasing the degree of information redundancy. This means that *it is possible to trade off 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, though. Addressing a weak failure semantics, able to span many failure behaviours, effectively translates in higher reliability—nevertheless,

(1) it **requires** large amounts of extra resources, and therefore implies a high cost penalty, and
(2) it **consumes** large amounts of extra resources, which translates into the 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 that is greater than the one required for tolerating, e.g., value failures. Using the simplest of the algorithms described in the cited paper, a 4-modular-redundant (4-MR) system can only withstand any *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 non-redundant 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 complexity of failure mode against number and type of faults tolerated* may be an important capability for an effective fault-tolerant structure. Dynamic adaptability to different environmental conditions[6] may provide a satisfactory answer to this need, especially when the additional complexity does not burden (and jeopardise) the application. Ideally, such complexity should be part of a custom architecture and not of the application. On the contrary, the 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 that complexity outside the user application may allow cost-effective verification, validation and testing. These processes may be unfeasible at application level.

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

3. CURRENT APPROACHES TO APPLICATION-LEVEL FAULT-TOLERANCE

One of the conclusions drawn in Sect. 1 is that the system to be made fault-tolerant must include provisions for fault-tolerance also in the application layer of a computer program. In that 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 methods, or approaches, 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 of them *with respect to the structural attributes defined in* Sect. 2.3 *and to various application domains.* A non-exhaustive list of the systems and projects implementing these approaches is also given. Conclusions are drawn in Sect. 4, where the need for more effective approaches is recognised.

[6]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".

Two of the above 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 Sect. 3.1. A third approach, described in Sect. 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 new custom high-level distributed programming languages or enhancing pre-existent languages of that kind. It is described in Sect. 3.3. A structuring technique called aspect-oriented programming, is reported in Sect. 3.4. Finally, Sect. 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**, i.e., N-fold ($N \geq 2$) replications in three domains:

Time. That is, repetition of computations.

Space. I.e., the adoption of multiple hardware channels (also called "lanes").

Information. That is, the adoption of multiple versions of software.

Following Avižienis [1985], it is possible to characterise 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]:

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

the meaning of which is "n executions, on m hardware channels, of p programs". The non-fault-tolerant system, or 1T/1H/1S, is called *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, e.g., 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, namely 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 in an increase of 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.

3.1.1.1 *The EFTOS library.*

EFTOS [De Florio et al. 1998a] (that is "Embedded, Fault-Tolerant Supercomputing") is the name of ESPRIT-IV project 21012.

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 has been 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 the adoption of the EFTOS library, the target embedded parallel application is plugged into a hierarchical, layered system whose structure and basic components (depicted in Fig. 1) are:

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

—One level upward (detection tool layer), a set of parametrisable 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, 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 send periodically "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 the components of the DIR net are combined into dependable mechanisms i.e., 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 "voting farm" [De Florio 1997b; De Florio et al. 1998a; 1998b], and so forth;

—The highest level (presentation layer) is given by a hypermedia distributed application which monitors the structure and the state of the user application [De Florio et al. 1998]. 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 in HTML [Berners-Lee and Connolly 1995], and remotely controls a Netscape browser [Zawinski 1994] so that it renders these HTML data.

3.1.1.2 *The SwiFT System.* SwiFT [Huang et al. 1996] stands for Software Implemented Fault-Tolerance. It includes a set of reusable software components (`watchd`, a general-purpose UNIX daemon watchdog timer; `libft`, a library of fault-tolerance methods, including single-version implementation of recovery blocks and N-version programming (see Sect. 3.1.2); `libckp`, i.e., a user-transparent

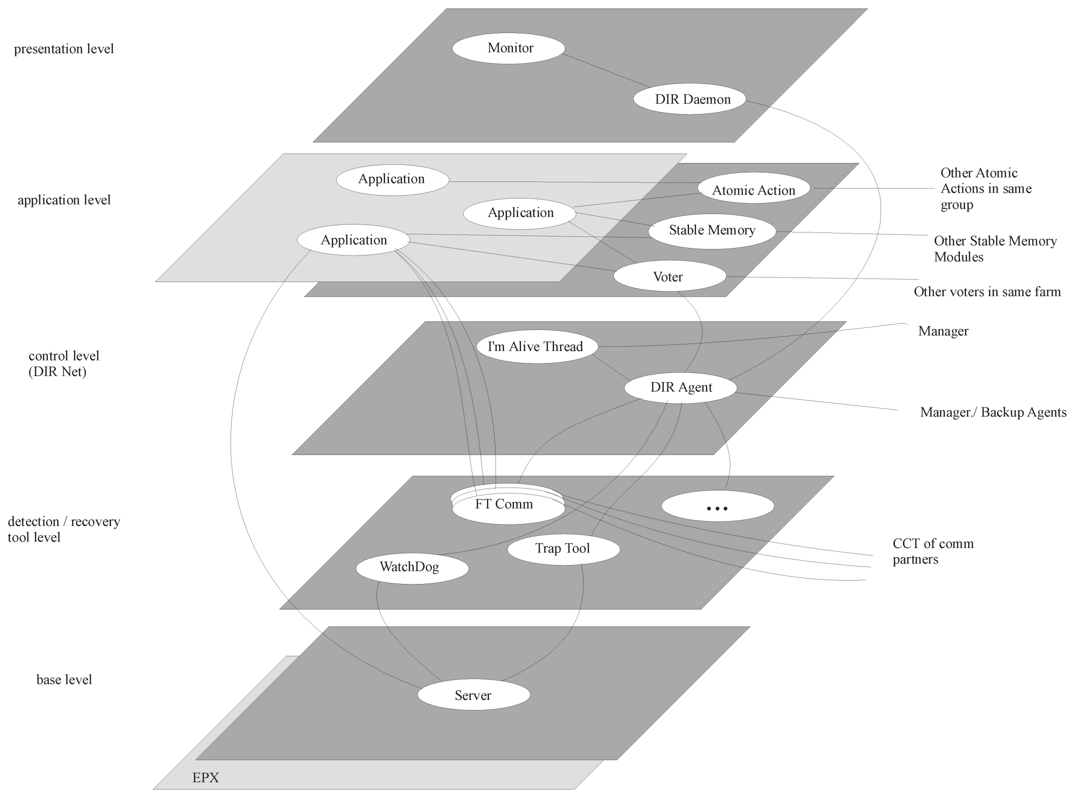


Fig. 1. The structure of the EFTOS library. Light gray has been used for the operating system and the user application, while dark gray layers pertain EFTOS.

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[7]. SwiFT has been successfully used and proved to be 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].

3.1.1.3 *Conclusions.* Figure 2 synthesizes the main characteristics of the SV approach: the functional and the fault-tolerance code 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 (hers) is the responsibility for doing it in a coher-

[7]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, be wiped out before they turn into a failure.

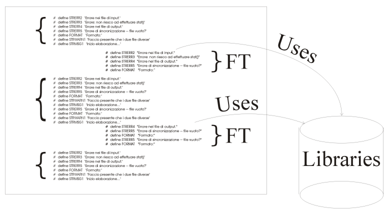


Fig. 2. A fault-tolerant program according to a SV system.

ent, effective, and efficient way. As it has been observed in Sect. 2.3, the resulting code is a mixture of functional code and of custom error-management code that does not always offer an acceptable degree of *portability* and *maintainability*. The functional and non-functional design concerns are not kept apart with 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 of syntactical adequacy (SA), we observe that following SV the fault-tolerance provisions are offered to the user though an interface based on a general-purpose language like 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 off-line and on-line 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 and with a good ratio of costs over improvements of the dependability attributes, sometimes introducing a relatively small overhead. Using these toolsets the designer can re-use existing, long tested, sophisticated pieces of software without having each time to "re-invent 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, a same service or functionality is supplied by N pieces of code that have been designed and developed by different, independent software teams[8]. The aim of this approach is to reduce the effects of design faults due to human mistakes committed at design time. The most used configurations are NT/1H/NS, i.e., N sequentially applicable alternate programs using the same hardware channel, and 1T/NH/NS, based on the parallel execution of the alternate programs on N, possibly diverse, hardware channels.

Two major approaches exist: the first one is known as recovery block [Randell

[8]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 4 briefly reports on the consequences of a well known case of redundant design.

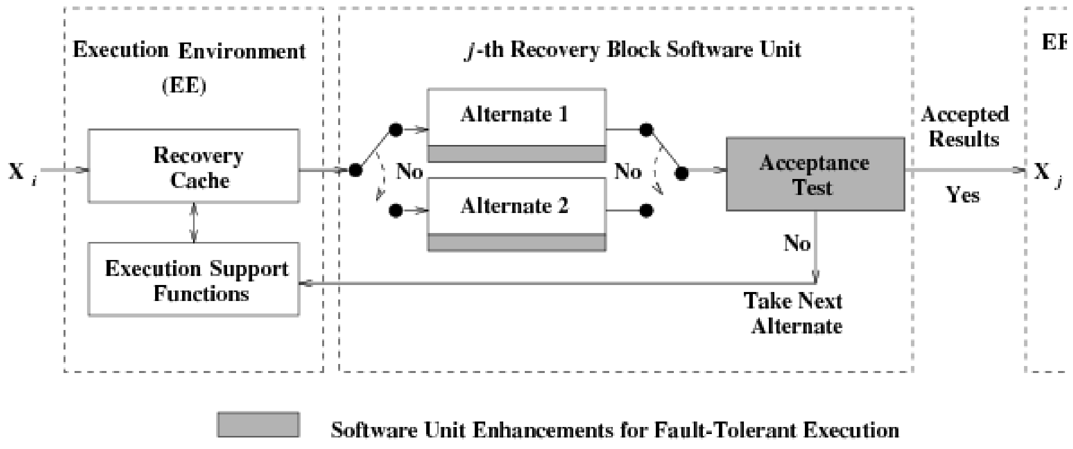

Execution Environment (EE)
j-th Recovery Block Software Unit
EE
X i
Recovery Cache
Execution Support Functions
Alternate 1
Alternate 2
No
No
Acceptance Test
Accepted Results
Yes
X j
No
Take Next Alternate
Software Unit Enhancements for Fault-Tolerant Execution

Fig. 3. The recovery block model with two alternates. The execution environment is charged with the management of the recovery cache and the 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.

1975; Randell and Xu 1995], and is dealt with in Sect. 3.1.2.1. The second approach is the so-called N-version programming [Avižienis 1985; 1995]. It is described in Sect. 3.1.2.2.

3.1.2.1 *The Recovery Block 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 "stand-by sparing" and described, e.g., in [Johnson 1989]. The approach is summarised in Fig. 3: 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 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 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 run-time 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 run-time overheads, must not retain data locally, and so forth. It must be regarded as the *ultimate* means for detecting errors, though not the *exclusive* one. Assertions and run-time 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 recovery blocks. This addition obviously further increases the complexity.

The SwiFT library that has been described in Sect. 3.1.1 (p. 11) 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 above mentioned requirements for enhancing the error detection coverage of the acceptance test. This would clearly require a run-time executive that is not part of this strategy. Other solutions, based on enhancing the grammar of pre-existing programming languages like Pascal [Shrivastava 1978] and Coral [Anderson et al. 1985], have some impact on portability. In both cases, code intrusion is not avoided. This translates in difficulties when trying to modify or maintain the application program without interfering "much" with the recovery structure, and vice-versa, when trying to modify or maintain the recovery structure without interfering "much" with the application program. Hence one can conclude that recovery blocks are characterised by unsatisfactory values of the structural attribute SC. Furthermore, a system structure for ALFT based exclusively on recovery blocks does not satisfy attribute SA[9]. Finally,

[9]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].

regarding attribute A, one can observe that recovery blocks are a rigid strategy that does not allow off-line configuration nor (*a fortiori*) code adaptability.

On the other hand, recovery blocks have been successfully adopted throughout 30 years in many different application fields. It has been successfully validated by a number of statistical experiments and through mathematical modelling [Randell and Xu 1995]. Its adoption as the sole fault-tolerance means, while developing complex applications, 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 monothonic function of x, y, z in any xT/yH/zS system.

3.1.2.2 **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, less often to class NT/1H/NS. 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 for being 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 Fig. 4) has been developed under the fundamental conjecture that independent designs translate in *random component failures*—i.e., statistical independence. Such a result would guarantee that correlated failures do not translate in immediate exhaustion of the available redundancy, as it would happen, e.g., using N copies of the same version. Replicating software would also mean replicating any dormant software fault in the source version—see, e.g., the accidents with the Therac-25 linear accelerator [Leveson 1995] or the Ariane 5 flight 502 [Inquiry Board Report 1996]. According to Avižienis, independent generation of the versions significantly reduces the probability of correlated failures. Unfortunately 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:

—Recovery blocks (in its original form) is 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[10].

[10]This weakness of NVP can be narrowed, if not solved, adopting the approach used in the so-called "voting farm" [De Florio et al. 1998b; 1998a; De Florio 1997b], a generic voting tool designed by one of the authors of this paper in the framework of his participation to project "EFTOS" (see Sect. 3.1.1.1): 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 may develop

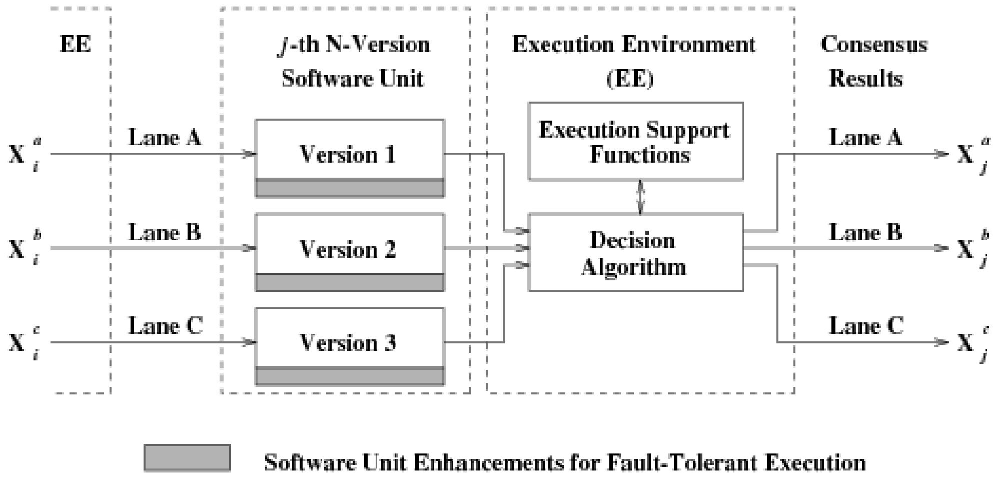

EE
j-th N-Version Software Unit
Execution Environment (EE)
Consensus Results
Lane A
Lane B
Lane C
Version 1
Version 2
Version 3
Execution Support Functions
Decision Algorithm
X_i^a
X_i^b
X_i^c
X_j^a
X_j^b
X_j^c
Software Unit Enhancements for Fault-Tolerant Execution

Fig. 4. 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".

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

3.1.2.3 *Conclusions.* As recovery blocks, also 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 according any programming paradigm. A generic, parametrisable architecture for real-time systems that supports straightforwardly the NVP strategy is GUARDS [Powell et al. 1999].

It is noteworthy to remark that the EE (also known as N-Version Executive) is a *complex* component that needs to manage a number of basic functions, for instance the execution of the decision algorithm, the assurance of input consistency for all versions, the inter-version communication, the version synchronisation and the enforcement of timing constraints [Avižienis 1995]. On the other hand, *this complexity is not part of the application software*—the N versions—that *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 Sect. 2.3, costs and times 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 re-

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—though is assisted in this simpler task by a number of template functions supplied with that tool.

covery 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.

Off-line adaptability to "bad" environments may be reached increasing the value of N—though this requires developing new versions—a costly activity for both times and costs. 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, as they need to be replicated and validated N times—as well as performed according to the NVP paradigm, so not to impact negatively on 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 Sect. 3.1.1, p. 11) 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 characterises NVP, as it requires some non-functional code to be included. This translates in 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 remark how the adoption of NVP as a system structure for ALFT requires a substantial increase in development and maintenance **costs**: both 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 payed back 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 Board Report 1996]. Nevertheless, the risks related to the chances of rapid exhaustion of redundancy due to a burst of correlated failures caused by a single or few design faults justify and call for the adoption of other

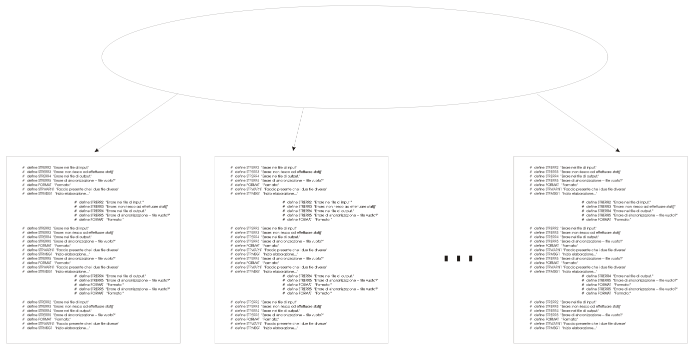


Fig. 5. A fault-tolerant program according to a MV system.

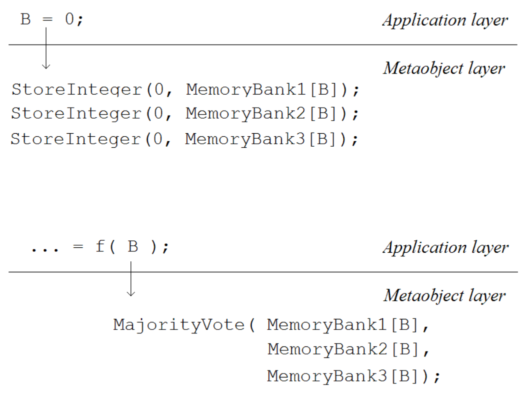
fault-tolerance provisions within and around the NVP unit in order to deal with the case of a failed NVP unit.

Figure 5 synthesizes the main characteristics of the MV approach: several replicas of (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, it does not produces very different output values when fed with slightly different inputs, data diversity may be a cost-effective way to fault-tolerance. Clearly in this case the voting mechanism does not run a simple majority voting but some vote fusion algorithm [Lorczak et al. 1989]. A typical application of data diversity is that of real time control programs, where sensor re-sampling 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 that of drastically decreasing design and maintenance costs, because 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 which allows 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 run-time executive of an object-oriented language like C++ or Java so that the developer can adopt and program different, custom semantics, adjusting the language to the needs of the user and to the requirements of the environment. Using MOPs, the programmer can modify the behaviour of fundamental



```
B = 0;
StoreInteger(0, MemoryBank1[B]);
StoreInteger(0, MemoryBank2[B]);
StoreInteger(0, MemoryBank3[B]);
```

```
... = f( B );
MajorityVote( MemoryBank1[B],
              MemoryBank2[B],
              MemoryBank3[B]);
```

Fig. 6. A MOP may be used to realize, e.g., triple-redundant memories in a fully transparent way.

features like 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, a MOP programmer may easily create "triple-redundant" memory cells to protect his/her variables against transient faults as depicted in Fig. 6.

The key concept behind MOPs is that of *computational reflection*, or the causal connection between a system and a meta-level description representing structural and computational aspects of that system [Maes 1987]. MOPs offer the metalevel programmer a representation of a system as a set of *metaobjects*, i.e., objects that represent and reflect properties of "real" objects, i.e., those objects that constitute the functional part of the user application. Metaobjects can for instance represent the structure of a class, or object interaction, or the 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 dynamical reorganisation of the structure and the operation of a system, e.g., 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 1996; 1998]. 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, synchronisation, voting... [Van Achteren 1997]) at meta-level. 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 Sect. 3.3.

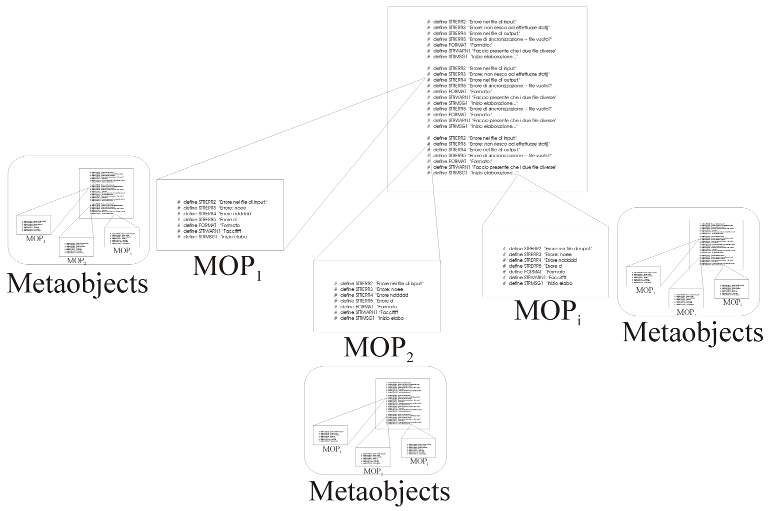


Fig. 7. A fault-tolerant program according to a MOP system.

3.2.1.1 *Conclusions.* MOPs are indeed a promising system structure for embedding different non-functional concerns in the application-level of a computer program. MOPs work at *language* level, providing means to modify the semantics of basic object-oriented language building blocks like object creation and deletion, calling and termination of class methods, and so forth. This appears to match perfectly to a proper subset of 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, i.e., optimal SC. Some other techniques, specifically those who might be described as "the most coarse-grained ones", such as distributed recovery blocks [Kim and Welch 1989], appear to be less suited for being efficiently implemented via MOPs. These techniques work at distributed, macroscopic level.

Figure 7 synthesizes the main characteristics of the MOP approach: The fault-tolerance programmer defines a number of metaobject protocols and associates them to 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 seem to constitute a promising technique for a transparent, coherent, and effective adoption of some of the 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 allows so far to estimate the practicality and the generality of this approach: "what 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 whether MOPs represent a practical solution towards the effective integration of most of the existing fault tolerance mechanisms in the user applications.

The above situation reminds the authors of another one, regarding the "quest" for a novel computational paradigm for parallel processing which would be capable of dealing effectively with the widest class of problems, like the Von Neumann paradigm does for sequential processing, though with the highest degree of efficiency and the least amount of changes 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," i.e., at macroscopic level, others were considering the problem exclusively at microscopical level. One can conclude that MOPs offer an elegant system structure to embed a set of non-functional 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 is remarked for instance in [Randell and Xu 1995; Lippert and Videira Lopes 2000]. It is therefore difficult to establish a qualitative assessment of attribute SA for MOPs.

The run-time management of libraries of MOPs 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 the 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 language level enhancing a pre-existing 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 Pre-existing Programming Languages.*

Enhancing a pre-existing programming language means augmenting the grammar of a wide-spread 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 systems based on this approach are presented: Arjuna, Sina, Linda, and FT-SR.

3.3.1.1 *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 done 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 the programmersrjuna stub generator attempts to compensate for these problems are 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.

to explicitly deal with tools to save and restore the state, to manage locks, and to declare in their applications instances of the class for managing atomic actions. As its authors state, in many respects Arjuna asks the programmer to be aware of several complexities—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.

3.3.1.2 *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 meta-level programming through the reification of messages. The concept of composition filters allows to implement several different "behaviours" corresponding to different non-functional concerns. SINA has been designed for being attached to existing languages: its first implementation, SINA/st, was for Smalltalk. It has been also implemented for C++ [Glandrup 1995]—the extended language has been called C++/CF. A preprocessor is used to translate a C++/CF source into standard C++ code.

3.3.1.3 *Fault-Tolerant Linda Systems.* The Linda [Carriero and Gelernter 1989b; 1989a] approach adopts a special model of communication, known as *generative communication* [Gelernter 1985]. According to this model, communication is still carried out through messages, though messages are not sent to one or more addressees, and eventually read by these latter—on the contrary, messages are included in a distributed (virtual) shared memory, called 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 non-blocking 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 synchronise. When more than one tuple matches a template, the choice of which actual tuple to address is done in a non-deterministic 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, a (no more 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, though Linda has been successfully employed in many different hardware architectures and many 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 its original form, Linda only offers single-op atomicity [Bakken and Schlichting 1995], i.e., 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 (via replicated state machines [Schneider 1990] kept consistent via ordered atomic multicast [Birman et al. 1991]) [Bakken and Schlichting 1995; Xu and Liskov 1989; Patterson et al. 1993], while others aim at combining multiple tuple-space operations into atomic transactions [Bakken and Schlichting 1995; Anderson and Shasha 1991; Cannon and Dunn 1992]. Other techniques have also been used, e.g., 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 that can be added to an existing programming language. The greater part of these extensions requires a preprocessor translating the extension in the host language. This is the case, e.g., 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

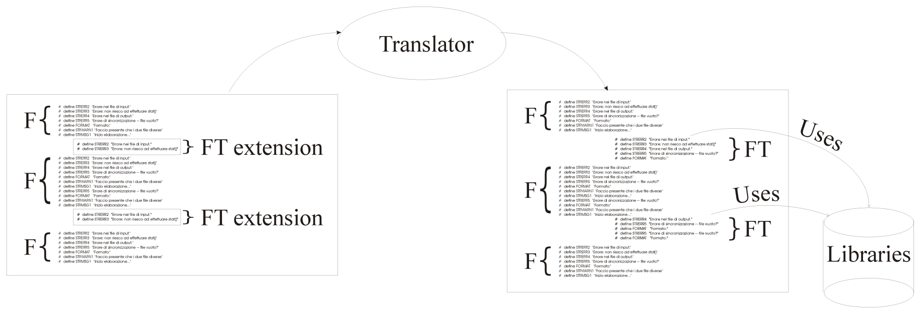


Fig. 8. A fault-tolerant program according to the enhanced language approach. Note how in this case a translator decomposes the program into a SV system.

is, e.g., 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.

3.3.1.4 *FT-SR.* FT-SR [Schlichting and Thomas 1995] 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 (FSM). A 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 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 to $n - 1$ failures. Whenever a failure exhausts the redundancy of a FSM, be that a simple or complex FSM, a failure notification is automatically sent to a number of other FSMs so 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 Sect. 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 to be false.

3.3.1.5 *Conclusions.* Figure 8 synthesizes the main characteristics of the enhanced language approach: A compiler or, as in the picture, a translator, produces a new fault-tolerant program. In the case in the picture the translated program belongs to class SV (see Sect. 3.1.1). Note how few clearly identifiable “FT” extensions are translated into larger sections of fault-tolerance code. As characteristics of SV system, this fault-tolerance code is undistinguishable from the functional code of the application.

We can conclude by stating that the approach of designing fault-tolerance enhancements for a pre-existing programming language does imply an *explicit code* *intrusion*: The extensions are designed with the explicit purpose to host a number of fault-tolerance provisions within the single source code. We observe, though, that being explicit, this code intrusion is such that the fault-tolerance 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 Fig. 8 we can conclude that the design and maintenance **costs** of this approach are in general less than those characterising SV.

On the contrary, the problem of hosting an adequate structure for ALFT can be complicated by the syntax constraints in the hosting language. This may prevent to incorporate a wide set of fault-tolerance provisions within a 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 pre-existing 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 characterised by good execution efficiency [Anderson et al. 1985; Schlichting and Thomas 1995].

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

As a final observation, we remark how the four cases that have been dealt with in Sect. 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 especially conceived to write fault-tolerant distributed software is discussed in the rest of this subsection.

3.3.2.1 *ARGUS.* Argus [Liskov 1988] is a distributed object-oriented programming language and operating system. Argus was designed to support application programs like banking systems. To capture the object-oriented nature of such programs, it provides a special kind 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 the one of transaction processing.

3.3.2.2 *The Correlate Language.* The Correlate object-oriented language [Robben 1999] adopts the concept of *active object*, defined as an object that has control over the synchronisation of incoming requests from other objects. Objects are active in the sense that they do not process immediately their requests—they may decide to delay a request until it is accepted, i.e., 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 servable, or until the object is destroyed. Conditions like an overflow in the request buffer are not dealt with in [Robben 1999]. If more than a single request becomes servable by an object, the choice is made non-deterministically. Correlate uses a communication model called "pattern-based group communication"—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 Sect. 3.3.1.3. Objects in Correlate are autonomous, in the sense that they may not only react to external stimuli but also give rise to autonomous operations motivated by an internal "goal". When invoking a method, the programmer can choose to block 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.

3.3.2.3 *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 research trend exists, which is based on the use of functional languages. This choice translates in a program structure that allows a straightforward inclusion of fault-tolerance means, 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 those mechanisms while maintaining the linguistic integrity of the adopted notation. in other words, optimising 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 synthesised attributes. Modules may refer to other modules. When modules do not refer any other module, they can be performed immediately. Such modules are called primitive modules. On the other hand, non-primitive 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 recursively this decomposition process 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 *computation tree*. This approach presents many benefits, e.g., as the order in which modules are decomposed is exclusively determined by attribute dependencies among submodules, a computation tree can be mapped onto a parallel processing means straightforwardly.

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 this features is called *redoing*. Redoing replaces a portion of the computation tree with a new computation. This is useful for instance

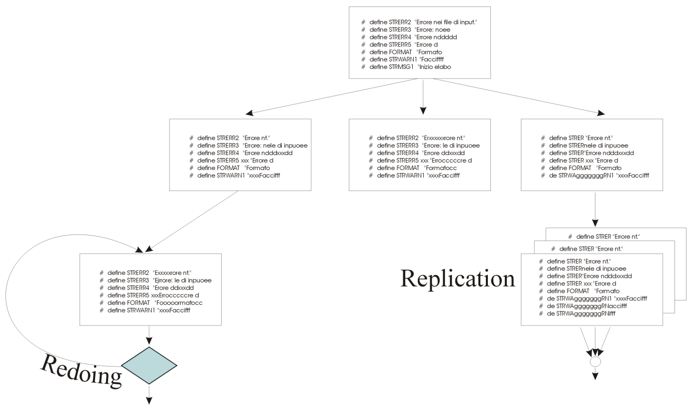


Fig. 9. A fault-tolerant program according to a FTAG system.

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 implement easily "retry blocks" and recovery blocks 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, e.g., 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 to allow the involved submodules to have the same set of inherited attributes and to generate a collated set of synthesised 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. Object versioning is also used, 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 how they are investigating the inclusion of other fault-tolerance features and trying to synthesise new expressive syntactical structures for FTAG—thus further improving attribute SA.

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

3.3.2.4 *Conclusions.* Synthesizing in a single meaningful picture the main characteristics of the approach that makes use of custom fault-tolerance languages is very difficult, for the design freedom translates in entities that may have few points in common. Figure 9 for instance synthesizes the characteristics of FTAG.

The ad hoc development of a fault-tolerance programming language allows in some cases to reach optimal values for attribute SA. The explicit, controlled intrusion of fault-tolerance code *explicitly encourages* the adoption of high-level fault-tolerance provisions and *requires dependability-aware design processes*, which translates in a positive assessment for attribute SC. On the contrary, with the same reasoning of Sect. 3.3.1, attribute A can be in general assessed as unsatisfactory[11].

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 know-how 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 a structuring technique that explicitly addresses, at 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 non-functional service within an application—for instance a system-wide error handling protocol—using either a procedural or an 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 those programming languages only provide abstraction and composition mechanisms to cleanly support the *functional* concerns. In other words, specific non-functional 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 non-functional services (the latter called "aspects" in AOP terms). These aspects tend to crosscut the system's class and module structure rather than staying, well localised, within one of these unit of functionality, e.g., a class. This increases the complexity of the resulting systems.

The main idea of AOP is to use:

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

(2) A so-called *aspect-oriented language* to implement given aspects by defining specific interconnections ("aspect programs" in AOP lingo) among the components in order to address various systemic concerns.

[11] In the case of FTAG, though, one could design a run-time 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.

(3) An *aspect weaver*, that takes as input both the aspect and the component programs and produces with those ("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, e.g., for a procedural language such as C. This package is only meant to be compiled and produce the ultimate executable code fulfilling a specific aspect like, e.g., higher dependability.

In a sense, AOP systematically automatises 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 those 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 generalisation of the ideas that are at the basis, for instance, of optimising compilers, program transformation systems, MOPs, and of 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 a "join points", i.e., 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 above). 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 pre-conditions, post-conditions, and invariants, that *determine how to use* and *what to expect* from a computational entity.

A study has been carried out on 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 Sect. 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

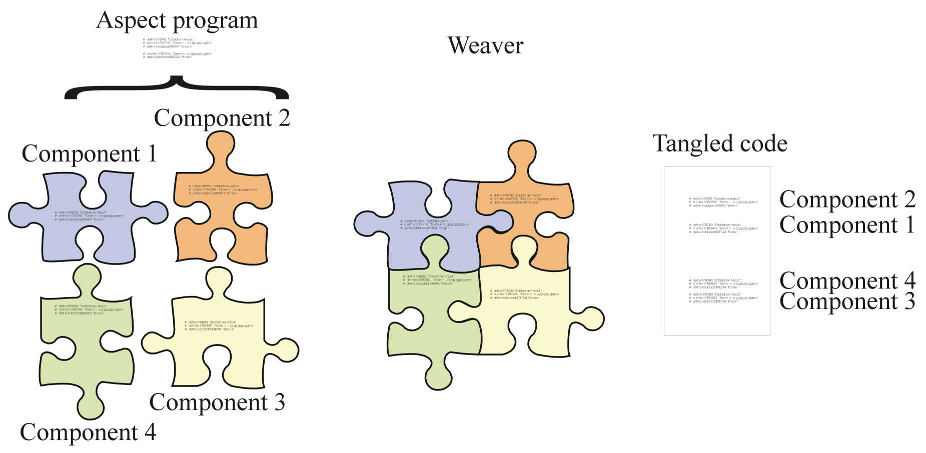


Fig. 10. A fault-tolerant program according to an AOP system.

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 dinamically over time, e.g., as a response to environmental changes. This translates in good support towards A.

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

3.4.3 *Conclusions.* Figure 10 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 re-engineering, which in principle should allow to reduce considerably **maintenance costs**.

AOP is a relatively recent approach to software development. AOP 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 payed back by the capability of addressing new aspects while keeping a single unmodified and general design.

For the time being it is not yet possible to tell whether AOP will spread out as a programming paradigm among academia and industry the way object-oriented programming has done since the Eighties. The many qualities of AOP are currently being quantitatively assessed, both with theoretical studies and with practical experience, and 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 the point of view of the dependability aspect, one can observe that AOP exhibits optimal SC ("by construction", in a sense [Kiczales and Mezini 2005]), and that recent results show that attribute A can in principle reach good values when making use of run-time weaving [Vasseur 2004], often realized by dynamic bytecode manipulation. This work by Ostermann [1999] is an interesting survey on this subject.

The adequacy at fulfilling attribute SA is indeed debatable also because, to date, no fault-tolerance aspect languages have been devised[12]—which may possibly be an interesting research domain.

3.5 The Recovery Meta-Program

The Recovery Meta-Program (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 a monitor, which:

—is scheduled when the application is stopped at some *breakpoints*,

—executes some sort of a *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 Sect. 3.1.2.1)—while programs are implementations of those strategies, e.g., of recovery blocks or N-version programming. The main benefit of RMP is in the fact that, while breakpoints require a (minimal) intervention of the functional-concerned programmer, RMP scripts can be designed and implemented without the intervention and even the awareness of the developer. In other words, RMP guarantees a good separation of design concerns. As an example, Fig. 11 shows how recovery blocks can be implemented in RMP:

—When the system encounters a breakpoint corresponding to the entrance of a recovery block, 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 meta-programs is Hoare's Communicating Sequential Processes language [Hoare 1978] (CSP).

[12]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 paper] lead

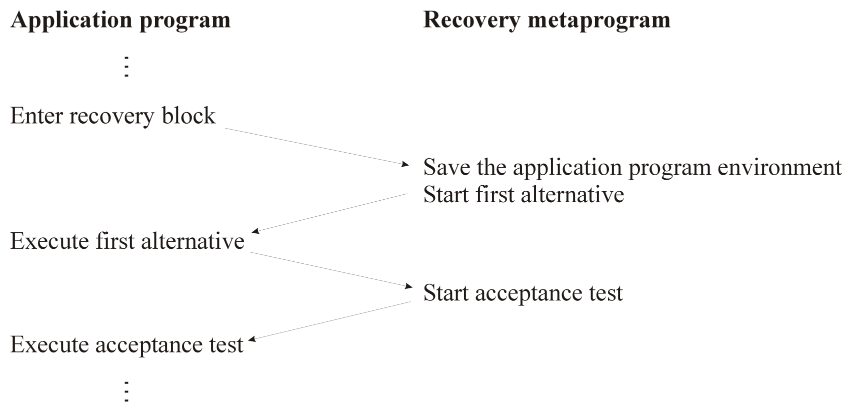
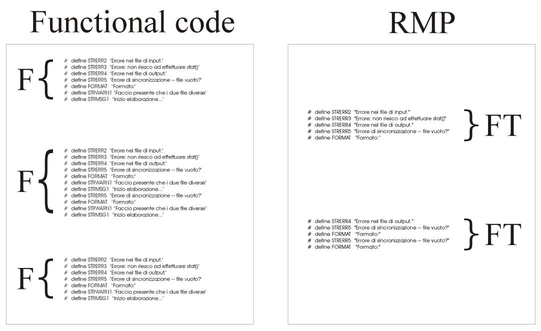


Fig. 11. Control flow between the application program and RMP while executing a fault-tolerance strategy based on recovery blocks.



```
#  define STRERR2  'Errore nei file di input.'
#  define STRERR3  'Errore: non riesco ad effettuare stat()'
#  define STRERR4  'Errore nel file di output.'
#  define STRERR5  'Errore di sincronizzazione -- file vuoto?'
#  define FORMAT   'Formato:'
#  define STRWARN1 'Faccio presente che i due file diverse'
#  define STRMSG1  'Inizio elaborazione...'

#  define STRERR2  'Errore nei file di input.'
#  define STRERR3  'Errore: non riesco ad effettuare stat()'
#  define STRERR4  'Errore nel file di output.'
#  define STRERR5  'Errore di sincronizzazione -- file vuoto?'
#  define FORMAT   'Formato:'
#  define STRWARN1 'Faccio presente che i due file diverse'
#  define STRMSG1  'Inizio elaborazione...'
#  define STRERR5  'Errore di sincronizzazione -- file vuoto?'
#  define FORMAT   'Formato:'
#  define STRWARN1 'Faccio presente che i due file diverse'
#  define STRMSG1  'Inizio elaborazione...'

#  define STRERR2  'Errore nei file di input.'
#  define STRERR3  'Errore: non riesco ad effettuare stat()'
#  define STRERR4  'Errore nel file di output.'
#  define STRERR5  'Errore di sincronizzazione -- file vuoto?'
#  define FORMAT   'Formato:'
#  define STRWARN1 'Faccio presente che i due file diverse'
#  define STRMSG1  'Inizio elaborazione...'
```

```
#  define STRERR2  "Errore nei file di input."
#  define STRERR3  "Errore: non riesco ad effettuare stat()"
#  define STRERR4  "Errore nel file di output."
#  define STRERR5  "Errore di sincronizzazione -- file vuoto?"
#  define FORMAT   "Formato:"

#  define STRERR4  "Errore nel file di output."
#  define STRERR5  "Errore di sincronizzazione -- file vuoto?"
#  define FORMAT   "Formato:"
#  define STRERR5  "Errore di sincronizzazione -- file vuoto?"
#  define FORMAT   "Formato:"
```

Fig. 12. A fault-tolerant program according to the RMP system.

3.6 Conclusions

Figure 8 synthesizes the main characteristics of RMP: The fault-tolerance code is in this case both logically and phisically distinct from the functional code, which means that the coding complexity and **costs** are considerably reduced.

In the RMP approach, all the technicalities related to the 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 characterised by optimal SC.

The design choice of using CSP to code the meta-programs influences negatively attribute SA. Choosing a pre-existent 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 pre-existing general-purpose distributed programming language as CSP is therefore questionable, as it appears

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".

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

Table I. A summary of the qualitative assessments proposed in Sect. 3. MV has been differentiated into recovery blocks (RB) and NVP. EL is the approach of Sect. 3.3.1, while DL is that of Sect. 3.3.2.

to be rather difficult or at least cumbersome to use it to express at least some of the fault-tolerance provisions. For instance, RMP proves to be 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 pre-existing language like CSP, RMP appears not to be the best choice for representing provisions such as, e.g., atomic actions [Jalote and Campbell 1985]. This translates in very limited SA.

Our conjecture is that the coexistence of two separate layers for the functional and the non-functional aspects could have been better exploited to reach the best of the two approaches: using a widespread programming language such as, e.g., C, for expressing the functional aspect, while devising a custom language for dealing with non-functional requirements, e.g., a language especially 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, e.g., the recovery metaprogram changes dynamically so as to compensate 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 to be characterised by a large overhead due to frequent context switching between the main application and the recovery metaprogram [Randell and Xu 1995]. Run-time requirements may be jeopardised by these large overheads, especially when it is difficult to establish time bounds for their extent. No other restrictions appear to be posed by RMP on the target application domain.

4. CONCLUSIONS

Five classes of system structures for ALFT have been described and critically reviewed, qualitatively, with respect to the structural attributes SC, SA, and A. Table I summarises the results of this survey providing a comparison of the various approaches. As can be seen from those 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.

The paper has also remarked the positive and negative issues of the evolutionary solution—using a pre-existing 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 how using an approach based on two languages—one covering the functional concerns and the other covering the fault-tolerance concerns—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 recently started European project ARFLEX ("Adaptive Robots for Flexible Manufacturing Systems").

ACKNOWLEDGMENT

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

REFERENCES

Aksit, M., Dijkstra, J., and Tripathi, A. 1991. Atomic delegation: Object-oriented transactions. *IEEE Software 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 Computer 23,* 10 (October), 23–32.

Anderson, B. G. and Shasha, D. 1991. Persistent Linda: Linda + transactions + query processing. In *Research Directions in High-Level Parallel Programming Languages*, J. Banâtre and D. Le Métayer, Eds. Number 574 in Lecture Notes in Computer Science. Springer, 93–109.

Anderson, T., Barrett, P., Halliwell, D., and Moulding, M. 1985. Software fault tolerance: an evaluation. *IEEE Trans. on Software Engineering 11,* 2, 1502–1510.

Andrews, G. R. and Olsson, L. L. 1993. *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings.

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

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

Bakken, D. E. and Schlichting, R. D. 1995. Supporting fault-tolerant parallel programming in Linda. *IEEE Trans. on Parallel and Distributed Systems 6,* 3 (March), 287–302.

Bao, Y., Sun, X., and Trivedi, K. 2003. Adaptive software rejuvenation: Degradation models and rejuvenation schemes. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN-2003).* IEEE Computer Society.

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

Berners-Lee, T. J. and Connolly, D. 1995. Hypertext markup language — 2.0. Tech. Rep. Request for Comments n. 1866, Network Working Group. Nov.

Birman, K., Schiper, A., and Stephenson, P. 1991. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems 9,* 3 (August).

Birrell, A. D. and Nelson, B. J. 1984. Implementing remote procedure calls. *ACM Trans. Computer Systems 2*, 39–59.

Bonér, J. 2004. Aspectwerkz - dynamic aop for java. In *Proceedings of AOSD 2004.*

Bonér, J. and Vasseur, A. 2004. Dynamic aop: Soa for the application. Tutorial presented at BEA eWorld 2004.

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 Comp. Surveys 21*, 323–357.

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

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

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

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

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 *Proc. of the 24th Euromicro Conference (Euromicro '98), Workshop on Dependable Computing Systems.* IEEE Comp. Soc. Press, Västerås, Sweden, 379–386.

DE FLORIO, V., DECONINCK, G., AND LAUWEREINS, R. 1998b. Software tool combining fault masking with user-defined recovery strategies. *IEE Proceedings – Software 145,* 6 (December), 203–211. Special Issue on Dependable Computing Systems. IEE in association with the British Computer Society.

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. 2000. An algorithm for tolerating crash failures in distributed systems. In *Proc. of the 7th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS).* IEEE Comp. Soc. Press, Edinburgh, Scotland, 9–17.

DE FLORIO, V., DECONINCK, G., LAUWEREINS, R., AND GRAEBER, S. 2001. Design and implementation of a data stabilizing software tool. In *Proc. of the 9th Euromicro Workshop on Parallel and Distributed Processing (Euro-PDP'01).* IEEE Comp. Soc. Press, Mantova, Italy.

DE FLORIO, V., DECONINCK, G., TRUYENS, M., ROSSEEL, W., AND LAUWEREINS, R. 1998. A hypermedia distributed application for monitoring and fault-injection in embedded fault-tolerant parallel programs. In *Proc. of the 6th Euromicro Workshop on Parallel and Distributed Processing (Euro-PDP'98).* IEEE Comp. Soc. Press, Madrid, Spain, 349–355.

DE FLORIO, V., MURGOLO, F. P., AND SPINELLI, V. 1994. PvmLinda: Integration of two different computation paradigms. In *Proc. of the First Euromicro Conference on Massively Parallel Computing Systems (MPCS'94).* IEEE Comp. Soc. Press, Ischia, Italy, 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 *Proc. of Specification and Verification of Component-Based Systems (SAVCBS'04), Workshop at ACM SIGSOFT 2004/FSE-12.*

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. on Software Engineering 17,* 7 (July), 692–702.

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

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 18,* 5 (Sept.-Oct.), 42–52. Special issue on Fault Tolerance.

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. 1996. Friends: A flexible architecture for implementing fault tolerant and secure applications. In *Proc. of the 2nd European Dependable Computing Conference (EDCC-2).* Taormina, Italy.

Fabre, J.-C. and Pérennou, T. 1998. A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach. *IEEE Transactions on Computers 47,* 1 (January), 78–95.

Gelernter, D. 1985. Generative communication in Linda. *ACM Trans. on Prog. Languages and Systems 7,* 1 (January).

Glandrup, M. H. J. 1995. Extending C++ using the concepts of composition filters. M.S. thesis, Dept. of Computer Science, University of Twente, NL.

Green, P. A. 1997. The art of creating reliable software-based systems using off-the-shelf software components. In *Proc. of the 16th Symposium on Reliable Distributed Systems (SRDS'97).* Durham, NC.

Guerraoui, R. and Schiper, A. 1997. Software-based replication for fault tolerance. *IEEE Computer 30,* 4 (April), 68–74.

Hansen, C. K., LaSala, K. P., Keene, S., and Coppola, A. 1999. The status of reliability engineering technology 1999 — a report to the IEEE Reliability Society. *Reliability Society Newsletter 45,* 1 (January), 10–14.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Comm. ACM 12,* 10 (October), 576–580.

Hoare, C. A. R. 1978. Communicating sequential processes. *Comm. ACM 21*, 667–677.

Horning, J. J. 1998. ACM Fellow Profile — James Jay (Jim) Horning. *ACM Software Engineering Notes 23,* 4 (July).

Huang, Y., Kintala, C., Kolettis, N., and Fulton, N. D. 1995. Software rejuvenation: Analysis, module and applications. In *Proc. of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25).*

Huang, Y. and Kintala, C. M. 1995. Software fault tolerance in the application layer. In *Software Fault Tolerance*, M. Lyu, Ed. John Wiley & 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 Technical Journal*, 29–37.

Inquiry Board Report. 1996. ARIANE 5 – flight 501 failure. Available at URL 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 1985).* IEEE Computer Society Press, Denver, Colorado. 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, Dept. of Computer Science, Oregon Graduate Institute.

Kiczales, G. 2000. AspectJ^{TM}: aspect-oriented programming using Java^{TM} technology. In *Proc. of the Sun's 2000 Worldwide Java Developer Conference (JavaOne).* San Francisco, California. Slides available at URL http://aspectj.org/servlets/AJSite?channel= documentation&subChannel=papersAndSlides.

Kiczales, G., des Rivières, J., and Bobrow, D. G. 1991. *The Art of the Metaobject Protocol.* The MIT Press, Cambridge, MA.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J. 1997. Aspect-oriented programming. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science.* Vol. 1241. Springer, Berlin, Finland.

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), Springer LNCS series.*

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. on Computers 38,* 5, 626–636.

Kleinrock, L. 1975. *Queueing Systems.* John Wiley & Sons. 2 volumes.

Knuth, D. E. 1984. Literate programming. *The Comp. Jour. 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.

Lamport, L., Shostak, R., and Pease, M. 1982. The Byzantine generals problem. *ACM Trans. on Programming Languages and Systems 4,* 3 (July), 384–401.

Laprie, J.-C. 1998. Dependability of computer systems: from concepts to limits. In *Proc. of the IFIP International Workshop on Dependable Computing and Its Applications (DCIA98).* Johannesburg, South Africa.

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 *Proc. of the 22nd International Conference on Software Engineering (ICSE'2000).* Limmerick, Ireland.

Liskov, B. 1988. Distributed programming in Argus. *Communications of the ACM 31,* 3 (March), 300–312.

Lorczak, P. R., Caglayan, A. K., and Eckhardt, D. E. 1989. A theoretical investigation of generalized voters for redundant systems. In *Proc. of the 19th Int. Symposium on Fault-Tolerant Computing (FTCS-19).* Chicago, IL, 444–451.

Lyu, M. 1995. *Software Fault Tolerance.* John Wiley & Sons, New York.

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

Lyu, M. R. 1998a. Design, testing, and evaluation techniques for software reliability engineering. In *Proc. of the 24th Euromicro Conf. on Engineering Systems and Software for the Next Decade (Euromicro'98), Workshop on Dependable Computing Systems.* IEEE Comp. Soc. Press, Västerås, Sweden, xxxix–xlvi. (Keynote speech).

Lyu, M. R. 1998b. Reliability-oriented software engineering: Design, testing and evaluation techniques. *IEE Proceedings – Software 145,* 6 (December), 191–197. Special Issue on Dependable Computing Systems.

Maes, P. 1987. Concepts and experiments in computational reflection. In *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA-87).* Orlando, FL, 147–155.

Masuhara, H., Matsuoka, S., Watanabe, T., and Yonezawa, A. 1992. Object-oriented concurrent reflective languages can be implemented efficiently. In *Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA-92).* 127–144.

Meyer, B. 1997. *Fault-Tolerant Computer Systems Design.* Prentice-Hall, New Jersey.

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

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

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 Computing Surveys 27,* 2 (June).

Parrington, G. D. 1990. Reliable distributed programming in c++: The arjuna approach. In *Proceedings of the Second 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 *Proc. of the 1993 Symposium on Applied Computing.* ACM/SIGAPP.

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. on Parallel and Distributed Systems 10,* 6 (June), 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. Software 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 & Sons, New York, Chapter 1, 1–21.

ROBBEN, B. 1999. Language technology and metalevel architectures for distributed objects. Ph.D. thesis, Dept. 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. on Computer Systems 2,* 4, 277–288.

SCHLICHTING, R. D. AND THOMAS, V. T. 1995. Programming language support for writing fault-tolerant distributed software. *IEEE Transactions on Computers 44,* 2 (February), 203–212. Special Issue on Fault-Tolerant Computing.

SCHNEIDER, F. 1990. Implementing fault-tolerant services using the state machine approach. *ACM Comp. Surveys 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 *Proc. 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. 1978. Sequential Pascal with recovery blocks. *Software — Practice and Experience 8*, 177–185.

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.

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

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. on Software Engineering 6,* 6 (November), 585–594.

VAN ACHTEREN, T. 1997. Object georienteerde afleiding van metaobjecten voor foutbestendigheid in de friends omgeving. M.S. thesis, Dept. 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 — The Journal of the American Ordnance Association.* Available at URL 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 *Proc. of the 19th Int. IEEE Symposium on Fault-Tolerant Computing (FTCS-19)*. IEEE, 199–206.

Zawinski, J. 1994. Remote control of UNIX Netscape. Tech. rep., Netscape Communications Corp. December. Available at URL http://home.netscape.com/newsref/std/x-remote.html.

Received September 2003; revised August 2005; accepted Month Year