Fault Tolerance Structures and Mechanisms for the GUARDS Architecture

A. Bondavalli*, S. Chiaradonna**, F. Di Giandomenico** and F. Grandoni**

* CNUCE Istituto del CNR, Via S. Maria 36, 56126 Pisa, Italy
a.bondavalli@cnuce.cnr.it

** IEI Istituto del CNR, Via S. Maria 46, 56126 Pisa, Italy
{digiandomanico, grandoni}@iei.pi.cnr.it

ABSTRACT

This report describes a set of fault tolerance organisations and identifies and discusses the basic supporting mechanisms that are needed for their usage in dependable systems for real-time critical applications implemented as instances of the GUARDS architecture.

1. INTRODUCTION

A rationale for the design of the GUARDS[1] architecture - Generic Upgradable Architecture for Real-time Dependable Systems - has been recently proposed in [Powell 1996]. It defines a multi-dimensional architecture where fault tolerance provisions can be developed at either the inter-channel or the intra-channel dimension or both. The architectural framework aims at defining a set of useful concepts and thus to put the basis for the definition of a family of possible instantiations which may differ from each other being tailored for specific end-user requirements. Actually, despite the common goal of all the possible instantiations of the GUARDS architecture is to support "dependable" applications, each individual instance will have its specific objectives in terms of the dependability attributes, and, more important will be subject to different constraints and assumptions originating from the specific surrounding environment.

In this report we will present and discuss a set of fault tolerant structuring solutions identifying supporting functionalities and mechanisms. The purpose is not to constitute a complete and consistent set of measures to be included in a single system, but to show solutions for different and sometimes contradictory goals towards the constitution of a (complete) shop-list, from which designers of specific instances of GUARDS systems may pick up what is necessary or appropriate for their system.

In the rest of this report we first recall some concept related to fault tolerance, then analyse and discuss the proposed general framework, highlighting the set of faults to be considered, the general assumptions and consequent constraints that must be considered for the identification, specification and design of the various fault tolerance structures. Fault tolerant structures aiming at error processing, to be used at the inter- and intra-channel dimension, are then identified, as well as fault treatment solutions for the proper management of transient faults. The intended behaviour of the proposed structures is described as C-like algorithms and the most important required mechanisms and functionalities are identified. At this stage we decided not to expand nor to detail the definition of the required identified supporting mechanisms to a kernel interface level. This is mainly due to 1) more discussions are needed and several issues of the architecture itself must be clarified (such as the computational models to be used), 2) no agreement has been reached so far on the supporting kernel interface. Further detailing will be performed once the set of fault tolerant structures will be agreed upon and decisions will be taken on which specification languages and kernel interfaces should be used.

2. OVERVIEW OF THE GUARDS ARCHITECTURE

2.1. FAULT -TOLERANCE

Fault tolerance is carried out by error processing and by fault treatment [Laprie 1995]. Error processing aims at removing errors from a computational state, if possible before any occurrences of failures; fault treatment aims at preventing faults from being activated again. Fault tolerance is a recursive concept: the mechanisms aimed at implementing fault tolerance should be protected against the faults which can affect them [Laprie 1995]. Examples are voters replication, self-checkers, "stable" memory for recovery programs and data.

Error processing may take on two forms: error recovery, where an error-free state is substituted for the erroneous state, or error compensation, where the erroneous state contains enough redundancy to enable the delivery of an error-free service from the erroneous (internal)

---

[1] GUARDS is the acronym for Generic Upgradable Architecture for Real-time Dependable Systems; it is an EC funded project under the IT framework (started on February 1996)

state. When error recovery is employed, the erroneous state needs to be (urgently) identified as being erroneous prior to being transformed, which is the purpose of error detection. On the other hand, compensation provides error masking.

Structured fault tolerance, i.e. techniques where redundancy (both for detection and correction) is applied to the individual components (hardware or software) with the goal of masking or detect errors internal to the component is the approach commonly used for error processing. Each technique has its own way of structuring the interactions among redundant parts and of managing the complexity added; it consists of a set of redundant components (either identical replicas or diversely implemented, functionally equivalent variants), plus adjudication subcomponents. Such modular-redundant designs differ in how they manage:

- error detection

- error correction (rollback vs voting vs other forward recovery)

- distribution of executions

- unconditional vs. on demand usage of redundancy

The two main approaches used for detecting errors (or accepting correct results), i.e. the adjudication, are **acceptance testing** (AT) to be performed on a single result or **"syntactic manipulations"** (mainly comparison) of the redundant results. AT's used at hardware layer are usually very general and as much as possible application independent, that is based on "syntactic" consistency checks of the results (e.g., range-checking on task inputs, divide-by-zero checks, etc.). In this respect, AT is the adjudication mechanism used for detecting errors in the safety net approach. AT's used at software layer are application based. Many forms of syntactic manipulations (comparison) are usually independent of semantics of the applications, although a wide variety of application specific adjudication or decision algorithms can be defined. In addition to detecting/correcting the values most applications require means to check correctness on the timing dimension, for these cases some kind watchdog timers are employed to prevent components from running for an inordinately long time.

Depending on how redundancy is utilised, the error processing schemes can be classified into two types: **static** structures that unconditionally consume a fixed amount of resources and adaptive or **dynamic** structures that use additional resources only when an error is detected. Static structures are intended to tolerate the maximum number of faults that may be present in the system and always execute all of their components regardless of the normal or abnormal state of the system; but, since such a "worst case" rarely happens, the amount of resources consumed is often higher than necessary. In this sense, they are not efficient. Dynamic structures, on the contrary, use their redundancy in a dynamic fashion only on occurrence of faults, in the hope that efficiency and performance will be improved. Of course, one must be aware that dynamic structures require extra hardware resources as well as extra time when faults occur and errors are detected. This behaviour has been demonstrated to offer better average response time and resource consumption at the price of higher variability. In addition it must be considered that dynamic structures need a more sophisticated control and introduce more complexity for the management of many system characteristics (such as communications, synchronisation, scheduling and compliance with real time requirements ...).

Obviously all these techniques aiming at providing correct results and able to detect/correct errors are not sufficient for assuring the proper functioning of systems. As faults accumulate in a system, these techniques are going to be defeated unless faulty components are isolated and reconfigurations take place.

Fault treatment consists of fault diagnosis and fault passivation. Fault diagnosis has the purpose of locating the source of the fault, i.e. the (hardware or software) component(s) affected, and understanding the nature of the fault (persistent, transient or intermittent) [Laprie 1995]. Fault passivation is carried out by removing or repairing the component identified as being faulty from further execution. No passivation is undertaken, if error processing directly

removes the fault, or if its likelihood of recurring is low enough. After reconfiguration the system may be no longer capable of delivering the same service as before, and may offer a degraded service.

A requirement which is clearly stated for GUARDS is the need (depending on the application field) to cope with temporary physical faults. Internal faults (usually termed intermittent) and external faults (known as transient) call for a specific treatment. Assuming all faults to be permanent, the steps leading from identification of the fault and location of the affected component to the passivation through a reconfiguration of the architecture are rather consolidated practice. Things are much less clear when temporary faults must be kept distinguished and specific treatment is sought. Actually for some GUARDS instance (e.g. those designed for railway interlocking or some instance of GUARDS for the nuclear submarine application) there are no major obstacles in handling temporary faults as permanent ones. After one occurrence of the fault, (especially if some spares are available) the faulty component may be passivated, extensively tested and replaced or put again in operation (if testing is passed successfully). If the time for 'repairing' is reasonably short, and the architecture is capable of dynamically reconfiguring while in operation, this solution may be the most beneficial. This procedure seems not to be viable for other application fields like other instances of the nuclear submarine application or satellites and space probes: in most cases no replacement is possible at all, the system must survive with the 'original' hardware components.

In these cases the system is required to treat temporary faults such that: i) the components must be kept in the system until 'certainty' has been achieved that the fault is permanent; ii) more (and more frequently occurring) faults have to be tolerated by the error processing mechanisms. This complicates the problem: it is no more sufficient to locate a component subject to a failure, but it is necessary to distinguish the type of fault the component is affected by. This additional information determines a greater delay in diagnosis, i.e. a longer fault latency, which must be accounted for by the error processing.

2.2. FAULTS

[Powell 1996] identifies the set of faults which GUARDS is intended to cope with, that is:

a) permanent and temporary / internal and external PHYSICAL FAULTS

b) permanent and temporary design faults

Not all the instances of GUARDS will have necessarily to deal with the complete set. For example some instance will be designed assuming the application software to be perfect (TA position at the GUARDS D1 meeting, Toulouse April 96), thus avoiding to include measures to tolerate software design faults.

Common mode failures: A fault tolerant design tries to avoid or limit as much as possible the causes of common mode failures among components (so that they can be assumed to fail independently) and to design structures such that individual failures of components may be tolerated by redundancy. Obviously this is very difficult to obtain.

As far as software is concerned the prevalent opinion in the community is that correlation cannot be avoided (without even mentioning the possibility that software without residual faults may ever exist) although some recent works report that a proper structuring of small modules and avoiding masking effects in the voting of small versions allowed to observe the absence of correlated failures in a specific individual experiment [Bishop 1995].

Failure independence is a simplifying assumption which is often less than adequate in redundant hardware systems too. In practice, attempts to estimate from field data the "beta factor" (probability that copy B fails, conditional on failure of copy A) in duplicated machinery often yield estimates greater than 0.1, and orders of magnitude greater than implied by assuming independence.

However this is one of the issues that must be addressed by the architecture definition and design. Addressing the permanent external physical faults [Powell 1996] writes: "...the GUARDS architecture should be able to ensure physical separation between redundant components".

Design faults: An interesting distinction is made in [Powell 1996] between temporary and permanent design faults. A temporary fault is defined as needing 'pointwise' conditions to occur whereas permanent design faults are 'systematic', meaning that they occur each time the system is started in the same conditions and the same set of inputs is provided. This distinction is not very clear from a logical point of view. All design faults are permanent (in the sense of systematic) if the set of inputs of a system is defined as the entire set of real world phenomena that affect its behaviour (thus including proper inputs, internal state of memories and registers, environmental conditions such as power, heat, magnetic interferences etc.). From a more practical (operational) point of view, one can think of inputs as constituted by those items that are under designers and users control: those explicitly defined as such (proper input values) reproducible internal states, physical solicitations that can be experimentally reproduced. This leads to and explains 'systematicity': if a fault is triggered exclusively by something that we can reproduce then we term it permanent and systematic, the other faults are such that are not under our control and possibility to investigate on in a systematic way.

Starting from these considerations it becomes clear that an asynchronous execution of several replicas have chances of not encountering all the 'pointwise' conditions necessary for triggering a fault. One could say that this asynchrony is the degree of diversity sufficient for tolerance. It is equally obvious that whenever controllable conditions (specific patterns of inputs, reproducible internal states) are alone sufficient to trigger design faults then a wider degree of diversity (from diverse hardware components to what is commonly termed as 'design' diversity) becomes necessary.

2.3. GUARDS ARCHITECTURE

The GUARDS physical architecture is a two-dimensional distributed architecture. Core of this architecture is the identification of the channels as the (physical) faults containment regions and the use of an inter-channel communication facility featuring interactive consistency capability to cope with arbitrary faults.

Fault tolerance may be provided by distributing the required redundancy:

a) at inter-channel level, where the same computation can be replicated and the set of results produced will be later processed by the output consolidation mechanism;

b) at intra-channel level, where redundancy confined inside an individual channel may be exploited for improving some dependability attributes on the service provided by the channel. Here the adjudication is made among values internal to the channel, where the possible spreading of fault effects makes irrelevant distribution or consistency issues.

Consequences:

1) the channel (from the system viewpoint) is the fault replacement unit;

2) the channel may or may not perform its activities exploiting internal redundancy;

3) the lack of the interactive consistency network inside a channel implies that the service provided by each channel can not be qualified with very high dependability figures; rather (according to the structure of the specific system, its output consolidation mechanisms and global requirements) it appears more interesting to employ solutions able to provide specific failure semantics (as fail silent or fail stop failure semantics) or improved availability or performance;

4) the channel itself (although a single failure containment region for the system) may contain several sub-regions (at least for some specific fault classes). So the fault assumption that can be made regarding certain specific (i.e. non COTS) components and the assumption of isolation among the various components wrt given faults becomes of paramount importance for the channel design (specifically for the design of the most appropriate fault tolerant structuring and for redundancy and reconfiguration management).

For a GUARDS instance consisting of 4 channels, or 3 channels plus interstages, (thus supporting the interactive consistency service) the simplest inter-channel fault tolerance structuring would be organised as follows:

- inputs are acquired by each channel and agreed upon through the consistency network
- each application is replicated on all the channels (either by simple replication or by diversity exploited at various degrees: from diverse hardware support to software diversity)
- adjudication is performed by means of the output consolidation mechanism.

Obviously this would be the simplest organisation reported as an extreme example suffering from many drawbacks and limitations.

More flexibility and the possibility to distinguish between applications of different criticality are offered if the FTPP approach [Harper and Lala 1990] is followed: redundancy is exploited by the definition of virtual processors. A virtual processor is defined as a set of processors each belonging to a different channel and on which the same computations are performed, while a physical processor may host at most one virtual processor. Thus fault tolerance is obtained only at the 'inter-channel' dimension while the processors belonging to the same channel are used for enhancing parallelism and performance. Following this approach in GUARDS, the consistency network service would be used for reaching agreement on the inputs and on intermediate results of the different applications, the adjudication could be realised through software replicated voters, thus allowing i) to reach agreement not just on outputs but also on internal information such as the system state, and ii) to implement application dependent adjudication strategies[2]. The output consolidation mechanisms (see ANSALDO 'exclusion logic' [Mongardi 1993]) may constitute in such a case the last check of consistency among the channels' output. Since virtual processors may be implemented by any number (from 1 to the number of channels) of physical processors, appropriate virtual processors may be specified according to the given application criticality level. Moreover, in this way physical separation (at least of the processors) of different integrity levels is achieved.

While FTPP exploits redundancy at the inter channel dimension, GUARDS considers the possibility to exploit redundancy at the intra-channel dimension as well. Instead of using a single processor in each channel as the component of the FTPP 'virtual processor', redundant structures may be considered, thus obtaining the 'two-dimensional' fault tolerance [Powell 1996]. Objectives of this second level of redundancy, confined inside an individual channel, must be different and should not overlap with those of the inter channel. Obviously the definition of redundant structures (even simple ones) in place of the single processor used by FTPP is going to add complexity with respect to at least scheduling strategies and their validation and synchronisation as required by the consistency network. On the other side, while inter channel redundancy seems to be limited to static replication (NMR style), intra-channel redundancy may accomodate a wider variety of solutions including usage of spares, time redundancy and combinations thereof.

A different view of inter- and intra- channel redundancy can and should be taken for those GUARDS instances consisting of just two channels. In these instances a specification diversity

---

2 a wide variety of adjudication mechanisms and algorithms can be defined [Di Giandomenico 90; Lorczak 1989; Arlat 96]

approach (like in ELEKTRA [Kantz and Koza 1995]) is possible. Actually no counter-indications are foreseen with respect to the use of this approach also in GUARDS instances with 3 or more channels. Specific functions in exclusive charge of the control of some actuators could be designed with this approach. The consequences on the entire instance will be to have specific output consolidation mechanisms for the interested actuators, while the computation could be accomodated by the definition of virtual processors involving a single channel (with redundancy at intra-channel level).

In the following we shall describe solutions both for the error processing and for the fault treatment. Based on the generic GUARDS architecture as drawn in [Powell 1996], error processing structures appropriate for the inter-channel level will be discussed separately from those pertinent to the intra-channel level, given the diverse implications associated with the employment of redundancy at these two levels. Then an approach for distinguishing between temporary and permanent faults will be described and possible applications to various parts of the GUARDS architecture suggested (consistency network, channel, individual components inside channels).

3. ERROR PROCESSING AT INTER-CHANNEL LEVEL

From [Powell 1996], instances of the GUARDS architecture may require a different number of channels, ranging from one single channel up to four. There, a description of the usage of the channel redundancy is given, for the cases of 2, 3 and 4 channels. The basic structure adopted is a parallel execution of the redundant components (the channels) in the style of NMR (with reference to hardware fault tolerance) [Siewiorek and Swarz 1982] or NVP (with reference to software fault tolerance) [Avizienis 1977], opportunely configured. Depending on the necessity to cope with design faults or not (both at software and at hardware level), diversely designed components or equal replicated components may be accomodated in this structure.

The behaviour of both NMR and NVP is basically the same; the only difference between the two error processing strategies is merely that NMR, being tailored for coping with hardware faults, employs identical replicas of the software modules, while NVP is generally meant to deal with software faults (especially, design faults) and thus diversely designed versions of the software modules are usually used. Given their similarity, in the description we provide in the following we mainly refer NMR (distinguishing among the three configurations 2MR, 3MR and 4MR, which are those of interest in the GUARDS context), although some considerations applying to NVP are also given when appropriate. This same organization is also applicable to treat the redundancy employed inside a channel to improve the dependability of the channel itself. The identification of approaches to error processing inside a channel is the topic of the next chapter; apart from specific considerations which will be pertinent to the inter-channel level only, the general description of the NMR given here in the following fully applies to the intra-channel level.

3.1. N-MODULAR REDUNDANCY (NMR)

The NMR strategy consists in the parallel execution of N redundant modules, acting on the same inputs, the outputs of which are collected by an adjudicator, responsible to select one single output. Employing all its redundancy at each execution, this strategy belongs to the static structures.

To exploit some degree of tolerance, at least 3 replicated components are necessary. Thus, being the GUARDS architecture limited to employ at most 4 channels, the two configurations 3MR and 4MR show tolerance to faults, while the 2MR only shows detection ability.

2MR is commonly used to satisfy fail-safe requirements: the double execution is followed by a comparison of the two results and any detected discrepancy leads the system to a safe state. Under the hypothesis that common mode failures cannot occur (which is usually acceptable when dealing with hardware fault tolerance, but not in the case of software fault tolerance, as

several studies proved [Knight and Leveson 1986]), this simple scheme always allows to detect a failure of one (or both) redundant modules.

The number of faults masked by the 3MR and 4MR configurations depends on the adjudication function adopted. Several algorithms for the adjudication are available from the literature; the choice of a suitable adjudicator depends on the characteristics of the application at hand. Usually, a majority voter is employed; in this case one channel fault can be tolerated by 3MR and 2 sequential channel faults by 4MR (or two channel faults during the same execution, provided that common mode failures cannot occur). A brief survey of commonly used adjudication functions already identified in [Arlat 1996] as interesting in GUARDS will be given later on.

After locating a failed channel (by some fault diagnosis mechanism), 3MR can be degraded to a 2MR configuration with detection capability only, while 4MR degrades to a 3MR configuration with still a degree of tolerance to faults.

Behaviour of NMR. The behaviour of NMR can be described by the following control algorithm in C-like language. Note that, although the description is given in terms of N redundant components in order to be general, in the GUARDS context N is expected to range from 2 to 4.

```
/* N :            number of redundant components*/
/* Mi :           i-th redundant component, i=1, 2,...,N */
/* Inp :          input values */
/* Ri :           result of i-th redundant component, i=1, 2,...,N */
/* Res :          adjudged result by the adjudication component */
/* State_mark :   current state of the execution of the scheme */

{
State_mark = failure;                    /* initialise the state variable */
parallel do
{                                        /* next operations executed in parallel */
 execute(M1, Inp, R1);                   /* execute component M1 */
  .....
 execute(MN, Inp, RN);                   /* execute component MN */
};                                       /* end parallel do */
adjudicate(R1, ...,RN, Res, State_mark);
        /* adjudicate the results Ri and set new state mark */
if (State_mark == E)
        /*E: successful state, i.e. the adjudged result can be delivered */
         deliver(Res);                   /* output Res as correct result */
else signal(failure);                    /* signal a detected failure */
}
```

Supporting mechanisms. Following the steps of the NMR execution, the mechanisms to be provided as services of the OS layer are:

1) **consistency of inputs (for parallel executions)**: to assure identical input values to each component executed in parallel. If the GUARDS instance employs a consistency network (as it would be in presence of 3 and 4 channels), this can be used to provide consistent input vectors to the channels. Note that in this case, if only two channels are interested at a given execution, input consistency cannot be assured, but the consistency check can help the correct channel(s) involved to understand a possible inconsistent behaviour shown by the other channel. Thus, no additional mechanism is required from the OS, unless those necessary to implement the consistency algorithm (for example, the buffering of the values to be circulated on the network) [Wellings 1996]. For instances which do not employ the consistency network (typically, 2 channels), the input consistency cannot be assured anymore; here, some form of low level communication mechanism between the two channels would be provided still preserving the semantic of failure containment region for the channel, and an exchange

of inputs can be performed before the two channels start executing, but without any guarantee that both will process the same input vector. When applied at intra-channel level, the mechanism to assure identical inputs to each redundant component should be always required from the OS, since no consistency network is available at this level.

2) **parallel execution**: to activate the execution in parallel of each single component of the whole redundant structure;

3) **output synchronisation**: necessary to allow: i) the consistency network to start an execution session on "homogeneous" output values (that is, those relating to the same redundant computation) and ii) to assure "homogeneous" adjudged results by each replicated adjudicator to the output data consolidation mechanism;

4) **adjudicators**: to select a single result from the several obtained by the redundant execution. A discussion on adjudication functions is conducted in the next subsection; here we anticipate that considerations on the relations often existing between adjudicators and the application semantics suggests that only simple adjudicators be implemented as OS services (for example, bit-by-bit voters to be employed in NMR structures to deal with hardware faults).

A note about the output data consolidation mechanism. Since the physical channels of an instance of the architecture can be grouped in different ways according to the dependability requirements of specific parts of the application running on them (for example, some parts need to be duplicated, others triplicated and so on), the output data consolidation mechanism should allow to manage a variable number of input values.

3.2 ADJUDICATION FUNCTIONS

Given specific application requirements, more or less complex adjudication functions can be thought to manage the outputs provided by the redundant channels execution. Although the specification of adjudication functions relate to the precise context in which the GUARDS instance is intended to operate, a set of general such algorithms can be identified from the literature e.g. [Lorczak 1989] which meet several different application dependability requirements. A more complete surwey is in [Di Giandomenico and Strigini 1990], where also the concept of optimal adjudication is developed and a set of characteristics and constraints for such an adudicator are identified. In [Arlat 1996], four commonly used decision techniques are suggested, extracted from [Lorczak 1989]. A brief description of them follows. Still many other decision algorithms may be derived from the optimal adjudicator developed in [Di Giandomenico and Strigini 1990], to better fit the specific requirements of GUARDS instances. The output data consolidation mechanism is not suitable to implement such functions, because of their relative complexity; moreover, being an instance of GUARDS endowed with only one output consolidation subsystem, it has to be general enough to accomodate for all applications needs. Therefore, adjudication functions should be software implemented; to avoid introducing single point of failure, they should be replicated on each channel. At the end of the redundant components execution, a consistent vector of the outputs produced is obtained by each channel through the consistency network; then each (correct) adjudication component replicated on the channels derives the same adjudged result from this output vector to forward to the output consolidation mechanism.

The four decision functions identified in [Arlat 1996] are:

- **the formalized majority voter** [Lorczak 1989]. This voter allows an inexact voting using a threshold ζ. Let R be the set of results to be adjudged, then the formalized majority voter starts with the selection of an output produced by the redundant execution, say x, and constructs the set of "good" outputs, say G, as the subset of R containing x and all the other values in R differing from x by less than ζ. If the values in G constitute at least a majority of all the values in R, then the final value is randomly selected from G. This algorithm is non-

deterministic: for a same set of output values, different choices of the initial output x may lead to find a set G or to terminate unsuccessfully;

- **the generalized median voter** [Lorczak 1989]. This voter extends the classical median function to non scalar-values, provided the output space is a metric space. The algorithm consists in repeatedly selecting the pair of outputs with maximum reciprocal distance and discarding them, until only one output is left, and using this as the adjudged output. It has been proved that an absolute majority of correct outputs no longer guarantees correct adjudication, unless all correct values coincide;

- **the formalized plurality voter** [Lorczak 1989]. It is a generalization of the formalized majority voter. It defines a partition of the set R such that each couple of outputs in each subset of this partition differ by at most ζ, and the subset can not be augmented with additional elements while preserving that property. Then, one of these subsets is chosen as the set G, if possible, for instance the one with the highest cardinality, or anyone that has cardinality higher than an assigned value. The final result is randomly selected from G. As for the formalized majority voter, this algorithm is non-deterministic: the achievement of a set G depends on the choice of the initial output value x chosen for the comparisons with the threshold ζ. So, for such adjudicators to produce a correct result, it is necessary that a set G with the required cardinality exists and that it is found; and, once it is found, a sufficient condition for the final result to be correct is that all the outputs values in G be correct. Unfortunately, no simple sufficient condition is known for this last property;

- **the weighted averaging voter** [Lorczak 1989]. This voter extends the classical mean adjudication function in the sense that the outputs participate to the definition of the mean value with different weights. The weights can be assigned in different ways, using additional information related to the trustworthiness of the redundant components, and known a-priori (and perhaps continually updated) or defined directly at invocation time (as, for example, by assigning results weights inversely proportional to their distances from all the other results). There is no guarantee that an upper bound on the number of incorrect results imply a correct adjudged result; however, this adjudication function seems well suited for situations where the probabilities of the values of the component outputs decrease with increasing distances from the ideal result (their errors in the terminology of physical measurements).

Is important to note that both the formalized majority voter and the formalized plurality voter are non deterministic algorithms. Since adjudication functions in GUARDS are expected to be replicated on each channel, it is necessary to refine the algorithms to eliminate any form of non-determinism, otherwise it cannot be assured that correct adjudicator replicas will make the same choice. This implies that, once the consolidated output vector is obtained, each adjudicator should start applying its algorithm on the same initial output value x. Since correct channels have identical consolidated output vector, it would be enough to fix in the algorithm the position of the consolidated output vector from which x have to be taken.

Moreover, the parameters involved in the above adjudication functions (the threshold ζ, the weights to assign to the output values) are in general strongly dependent on the application, so that values suitable for generic use can not be determined; the application programmer should therefore identify significant values for them. This would imply that adjudicators are better implemented at the application level and not merely furnished as mechanisms of the OS layer, although simple forms (like voters to employ in hardware fault tolerance structures) could be part of the OS services.

3.3 SAFETY CHECKING (SPECIFICATION DIVERSITY)

An alternative to the organization of a 2 channel redundancy as suggested in [Powell 1996] in order to satisfy safety requirements is the approach followed in the ELEKTRA railway signalling system [Kantz and Koza 1995]. The ELEKTRA architecture employs 2 channels to

realize specification diversity: one channel executes the "real" interlocking function, while the other one checks that the operations performed by the former do not violate the imposed safety conditions. So, the first channel is implemented according to a functional specification, while the second is developed on the basis of specifications derived from the operating regulations in the field. Actions are only performed if the second channel agrees on the results produced by the first channel, otherwise the system transits to a safe state. In addition, each channel executes acceptance tests on exchanged results to enhance the self-checking property of the channel itself. The replication in a TMR style inside each channel is exploited in order to enhance the availability and reliability of the channel itself. The considerations which convinced the authors to choose this approach instead of more classical fault tolerance techniques based on design diversity are that in the latter: i) common mode failures in the implementation constitute a quite hard problem to deal with; ii) faults in the specification are not tolerated.

The implementation of specification diversity is highly application dependent. In fact, the monitoring of the activities performed by the "functional" channel implies the knowledge/implementation of those safety properties regulating that specific functional specification. The output consolidation mechanism needs added flexibility: in fact, specification diversity applied at inter-channel level introduces an asymmetry in the channel outputs. Typically, the monitor channel output has to be used to gate the output produced by the functional channel to the external subsystem; of course, system design must assure that a malfunction in the monitor channel, unduly blocking a correct functional command, should result into safe outputs. If this channel redundancy configuration is to be supported by the GUARDS architecture, the output consolidation mechanism should exhibit the described functionality. Particular care should be devoted to the communication layer between the two channels, as outlined in [Kantz and Koza 1995]. In case this organization be employed in a GUARDS instance equipped with the interactive consistency network, then a reliable means is directly provided to exchange data between the two channels.

4. ERROR PROCESSING AT INTRA-CHANNEL LEVEL

In GUARDS, redundancy in channel resources can be devoted to increase performance, availability, or both [Powell 1996]. In fact, end-user requirements (executive board meeting, Toulouse July 96) call for channel reliability figures in the order of 100 times that of COTS components. Therefore, the generic architecture must cater for redundant structures at the channel level. The cited figures justify the adoption of rather simple, albeit well proven, schemes, which are supplemented here by some more sophisticated artefacts, will some user be wary to exploit the best performance/cost ratio they allow.

In the redundant structures sketched in the sequel, bear in mind that the number N of redundant components can be considered in the range of 2 to 4 for reasonable GUARDS instances.

4.1. STANDBY SPARING

Adjudication is made by acceptance tests. One component is executed, at first, and if its result does not pass an acceptance test, other components are invoked (using the same input data), in turn, until one passes or all the available components fail. In the latter case, a catastrophic failure occurs. This structure with $N \geq 2$ aims at tolerating N - 1 faults. When the redundant components are software variants this technique is known as recovery blocks (RB), the first scheme designed to provide software fault tolerance [Randell 1975]. The main drawbacks of this simple structure are i) the difficulty of providing high-confidence acceptance tests, and ii) the large time overhead incurred in case of fault.

In this approach, the acceptance test is applied sequentially to the results produced by the components. The redundancy is thus used in a dynamic fashion being exploited on occurrence of faults. The execution time of this scheme is normally that of the first component, acceptance test, and the operations required to establish and discard a checkpoint. This will not impose a high run-time overhead unless an error is detected and backward recovery required. In this

regard, standby sparing is highly efficient. On the other side, this organisation may suffer from relatively long response time, especially in the worst case.

Standby sparing in itself aims at tolerating faults, and no redundant components, other than the acceptance test, are necessary if fault detection is the objective. A typical way of obtaining structures with the fail stop semantics is building self-checking structures where a component performing the application chore is associated to an acceptance test performing validity checks.

Behaviour of standby sparing. The behaviour of standby sparing can be described by the control algorithm in C-like language, below.

```
/* N :          number of redundant components */
/* Mi :         i-th redundant component, i=1, 2,...,N */
/* Inp :        input values */
/* Res :        result of the current component */
/* State_mark : current state of the execution of the scheme */

{
 State_mark = failure;                          /* current state variable initialisation */
 i = 0;                                         /* set index of the current component */
 do                                             /* repeat the cycle */
 {
  i= i +1;                                      /* start next execution */
  execute(Mi, Inp, Res);                        /* execute active component */
   acceptance_test(Res, State_mark);
   /*execute the AT on the result of current component and set new state mark */
 } while (State_mark <> E and i < N);
                  /* E : successful state, i.e. result pass the acceptance test */
 if (State_mark == E)
        deliver(Res);                           /* output Res as correct result */
 else signal(failure);                          /* signal a detected failure */
}
```

Supporting mechanisms. In the following, the mechanisms to be provided as services of the OS layer for the standby sparing structure are identified.

1) **input persistency**: to store the initial (internal) state to assure identical input values for later execution by other components;

2) **acceptance tests**: to detect errors and produce a judgement on each redundant component executed (being performed on the single results of each redundant component executed).

4.2. N-SELF-CHECKING

Another classic idea is to combine together components with a fail-stop semantics to improve availability. In this approach, fault tolerance is provided through active dynamic redundancy based on a number of self-checking components (as in the ATT ESSs, Stratus systems, etc.). Each self-checking component may be constructed as: i) two components plus a comparator or ii) a single one associated with an acceptance test. One component is regarded as the active one (in charge of delivering the service), and the others as "hot" standby spares. Upon failure of the active component, service delivery is switched to a "hot" spare. Usually, the execution is parallel (space redundancy) and fixed redundancy is used, independently of fault occurrences. This technique applied to software is known as N-self-checking programming (NSCP) [Laprie et al. 1987].

Through the parallel execution of $N \geq 2$ self-checking components, tolerance of k simultaneous faults in distinct self-checking components requires to employ $N = k+1$ self-checking components.

This technique assumes components with a self-checking failure semantics, thus no structuring is needed to provide detection.

The cost of the duplication for constructing self-checking components may be very high [Rennels 1984], therefore this approach does not seem to be appropriate if the goal is to trade between reliability and efficiency; on the other side its usage may bring good results to improve availability.

Behaviour of NSC. The behaviour of N-self-checking can be described by this control algorithm in C-like language, with comments on the right side.

```
/* N :              number of redundant self-checking components */
/* SCi :            i-th self-checking component, i=1, 2,...,N */
/* Inp :            input values */
/* Ri :             result of i-th self-checking component, if exists, i=1, 2,...,N */
/* State_marki :    current state of the execution of i-th component SCi */
/* i :              index of the active self-checking component  */
{
 for(j=1; j<=N; j++) State_markj = failure;   /*initialize the current state*/
  parallel do
 {                                            /* next operations executed in parallel */
 execute(SC1, Inp, R1, State_mark1);          /* execute SC1 and set new state mark */
  .....
 execute(SCN, Inp, RN, State_markN);          /* execute SCN and set new state mark */
};                                            /* end parallel do */
 for(j=1; State_markj <> E and j<N; j++)      /* E : successfull state */
  i= (i mod N)+1;                    /* i-th component fails, service delivery is switched */
 if (State_markj == E)
        deliver(Ri);                          /* output Ri as correct result */
 else signal(failure);                        /* signal a detected failure */
}
```

Supporting mechanisms.

1) **consistence of inputs (for parallel executions)**: to assure identical input values to each self-checking component executed in parallel;

2) **parallel execution**: to activate the execution in parallel of each single self-checking component of the whole redundant structure; in this embodiment, the N-self-checking scheme is implemented with a synchronization at the end of the component execution. This condition can be relaxed to gain in throughput, at the cost of increased complexity in the redundancy management;

3) **switching service delivery**: upon the active component failure, the actual service delivery has to be switched to a "hot" spare; the latter becomes the new active component.

A self-checking component may be an off-the-shelf product, or may be built using two standard processors plus a comparator. In the latter case, two other mechanisms are required:

4) **output synchronization**: necessary on the outputs produced by two (sub)components to perform the required comparison;

5) **comparators of two results**: to produce a judgement on the self-checking component;

A self-checking component may be also constructed by using a single component associated with an acceptance test; in this case, only one other mechanism is required:

6) **acceptance tests**: to detect errors and produce a judgement on the self-checking component.

4.3. SELF-CONFIGURING OPTIMAL PROGRAMMING (SCOP)

N-Modular-Redundancy and Standby-sparing represent two extremes on many interesting dimensions of fault tolerance structures: parallel vs. sequential execution, static vs. dynamic usage of redundancy, comparison-based vs. acceptance tests checking of results. Obviously many interesting trade offs may be devised and have been proposed. An interesting method for describing families of such intermediate solutions, which is also general enough to encompass the two extremes, is represented by SCOP [Xu et al. 1995].

Basic Description of SCOP. The SCOP scheme, initially devised for software components only, consists of a set of components $M=\{M_1, M_2, ..., M_N\}$, an adjudication mechanism, a set of delivery conditions, one of which is to be dynamically chosen at run time, and a controller that coordinates dynamic actions of the architecture. At run time an instance of SCOP accepts as additional parameters the selected delivery condition and, possibly, a deadline for the whole execution. The controller decides first how many phases can be performed (in order to provide a timely result), then, at each phase, it selects the (minimum) set of components that (if successful) could satisfy the selected delivery condition. Upon execution of the currently active set of components, the adjudicator verifies if the chosen delivery condition has been met (using the collection of results - *syndrome* - that grows as more phases are performed). This behaviour is repeated until a result can be delivered or the components are exhausted.

An instance of SCOP can be designed to obey multiple different delivery conditions. One of them is dynamically chosen at run time, and the selected condition may change for different executions, according to the degradation of the system. In addition, if SCOP is used for the provision of a service used by many different applications, different delivery conditions may be dynamically chosen by the different applications, according to their degrees of criticality. Since the different conditions will usually have different fault coverage, SCOP is therefore able to provide different levels of dependability.

The scheme is very general allowing to combine several approaches for masking the effect of faults with different delivery conditions. For example, combining the design diversity approach with an acceptance test the Recovery block behaviour is obtained, while a pure replication and a majority voter (with the selection of one phase only) can be used for the design of an instance of NMR. This way the best alternative appropriate for the specific application can be specified and designed.

For the sake of simplicity, two SCOP configurations are here proposed in restricted versions: SCOP2+1, where two components are executed in the first phase, and one more in the second if the delivery condition was not met; SCOP2+2, where in the second phase a new couple of components is run.

Behaviour of SCOP2+1. The behaviour of SCOP2+1 can be described more precisely by the following control algorithm.

```
/* {M1, M2} :      currently active set (CAS) at the first phase */
/* {M3} :          currently active set (CAS) at the second phase */
/* Inp :           input values */
/* Ri :            result of i-th redundant component, i=1, 2, 3 */
/* Res :           adjudged result at each phase */
/* State_mark :    current state of the execution of the scheme */

{
State_mark = NE;                    /* initialise current state as non end-state */
parallel do
{                                   /* next operations executed in parallel */
 execute(M1, Inp, R1);              /* execute component M1 */
 execute(M2, Inp, R2);              /* execute component M2 */
}
```

```
compare(R1, R2, Res, State_mark);
if (State_mark == E)                    /*E : successful state, i.e. result can be delivered */
        deliver(Res);                   /* output Res as correct result */
else
 {                                      /* start second phase */
        execute(M3, Inp, R3);           /* execute component M3 */
        adjudicate(R1, R2, R3, Res, State_mark);
        if (State_mark == E)
                deliver(Res);           /* output Res as correct result */
        else signal(failure);           /* signal a detected failure */
 }
}
```

Behaviour of SCOP2+2. The behaviour of SCOP2+2 can be described more precisely by the following control algorithm with comments on the right side.

```
/* {M1, M2} :        currently active set (CAS) at the first phase */
/* {M3, M4} :        currently active set (CAS) at the second phase */
/* Inp :             input values */
/* Ri :              result of i-th redundant component, i=1, 2, 3, 4 */
/* Res :             adjudged result by the adjudication component at each phase */
/* State_mark :      current state of the execution of the scheme */

{
State_mark = NE;                        /* initialise current state as non end-state */
parallel do
{                                       /* next operations executed in parallel */
 execute(M1, Inp, R1);                  /* execute component M1 */
 execute(M2, Inp, R2);                  /* execute component M2 */
}                                       /* end parallel do */
compare(R1, R2, Res, State_mark);
if (State_mark == E)                    /*E : successful state, i.e. result can be delivered */
        deliver(Res);                   /* output Res as correct result */
else
 {                                      /* start second phase */
        parallel do
        {                               /* next operations executed in parallel */
         execute(M3, Inp, R3);          /* execute component M3 */
         execute(M4, Inp, R4);          /* execute component M4 */
        };                              /* end parallel do */
        adjudicate(R1, R2, R3, R4, Res, State_mark);
        if (State_mark == E)
                deliver(Res);           /* output Res as correct result */
        else signal(failure);           /* signal a detected failure */
 }
}
```

Supporting mechanisms. The mechanisms to be provided as services of the OS layer to implement the SCOP structure are identified.

1) **consistence of inputs (for parallel or sequential executions)**: to assure identical input values to each component executed in parallel or in sequence;

2) **parallel execution**: to activate, in different phases, the parallel execution of a sub-set of components of the whole redundant structure;

3) **output synchronization**: necessary on the outputs produced by the components executed in each phase to perform their comparison;

4) **adjudicators**: to select the result of the redundant computation; this function degenerates to simple comparison in some instance. In the general case, the adjudication in SCOP is incremental in nature; this requires buffering of previous phase's results. Inexact voting can be adopted (see for examples Section 3.2).

5. FAULT TREATMENT (AND MANAGEMENT OF TRANSIENT FAULTS)

As already mentioned in [Powell 1996], for high-availability applications it is necessary to perform fault diagnosis, fault passivation and possibly system reconfiguration. In fact, without any action devoted to the treatment of faults, the only error processing mechanisms cannot take into life the system for too long (unless very high redundancy is employed, which is usually contrasting with other system requirements, like costs and size of the whole system). The presence of temporary faults, which, as already stated, represent a large majority of faults experienced by current computing systems, would impose that the fault diagnosis algorithm be able to discriminate the kind of persistence of the fault affecting the system components, in order to take appropriate actions.

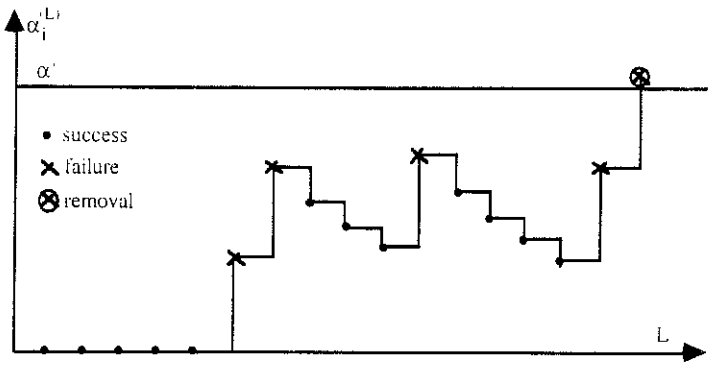
Although not formalized and explicitly introduced as mechanisms devoted to fault diagnoses, common-sense rules are sometimes employed in existing systems to realize naive means, associated to well defined error processing structures, to cope with transient faults. Examples can be found in the architecture proposed in [Mongardi 1993], where two consecutive failures experienced by the same hardware component being part of a redundant structure make the other redundant components to consider it as definitively faulty. In [Lala and Alger 88] if, after a fault, a channel of the core FTP can be restarted successfully (restoring its internal state from other operating channels), then it is brought back in operation; "however, it is assigned a demerit in its dynamic health variable; this variable is used to differentiate between transient and intermittent failures". As a third example, in [Agrawal 88] a list of 'suspected' processors is generated during the redundant executions; then, a few schemes are suggested for processing this list, from taking the processors in it down for off-line diagnosis to assigning weights to processors participating in the execution of a job and failing to produce a matching signature with that of the accepted result and taking down for diagnostics those whose weight exceeds a certain predetermined threshold.

A fault treatment structure is proposed here, to be employed in conjunction with an error processing provision, with the following highlights: i) the adopted fault model includes permanent, intermittent and transient faults; ii) specific effort is made for the fault treatment mechanism to avoid removing from service components experiencing transient faults; iii) the complete fault tolerance strategy pays specific attention to an efficient use of available resources while improving the dependability. The structure adheres to a very simple model: this allows easy and extensive evaluation of its behaviour by means of standard tools; because of this feature it appears to be appealing to the GUARDS architecture. The idea is that, since error processing techniques are employed to pick up a correct computation result, to be delivered outside, the information gathered in this phase can be used also to signal the erroneous behaviour of components to the fault-treatment mechanism. The error processing schemes can be exploited in this way only if they give, as a side effect, unambiguous information on the identity of misbehaving components; as a negative example, the generalized median voter (see Section 3.2) is not amenable to this purpose. A short description of this mechanism, which will be called *α–count*, follows.

The judgement issued by error processing is symbolically expressed as a binary value. Let $J_i^{(L)}$ indicate the L-th judgement on the generic component u_i; then $J_i^{(L)} = 0$ means success while $J_i^{(L)} = 1$ means failure. Each judgement is correct with a probability, which depends on the error processing. The α-count keeps track of fault occurrences in each component while execution proceeds. A score α_i is associated to each not-yet-removed component u_i to record information about the failures experienced by that component. α_i is initially set to 0, and accounts for the pertinent L-th judgement as follows:

$$\alpha_i^{(L)} = \begin{cases} \alpha_i^{(L-1)} \cdot K & \text{if } J_i^{(L)} = 0 \\ \alpha_i^{(L-1)} + 1 & \text{if } J_i^{(L)} = 1 \end{cases} \qquad K \le 1$$

When the value of $\alpha_i^{(L)}$ grows bigger than or equal to a given threshold α', the component u_i is diagnosed as faulty: this event is signalled to the fault passivation mechanism. The combined effects of the eventual removal upon system performance and reliability figures depend on the parameters K and α'. α' represents the minimum number of consecutive failures sufficient to consider a component permanently faulty, while K represents the ratio by which α_i is decreased after a success. The following figure gives a qualitative representation of the evolution of $\alpha_i^{(L)}$.



Qualitative description of $\alpha_i^{(L)}$

The behaviour of the α-count can be modeled e.g. with Stochastic Activity Nets (SAN); the fundamental simplicity of the mechanism results into simple models. Therefore, the mechanism can be easily analyzed and evaluated with the help of automatic tools, such as SURF-2 or UltraSAN.

The performance of the α-count-based fault-treatment, that is its capability to diagnose faulty components as soon as possible and to lower the probability of removing non-faulty components, depends on the relative percentage of permanent, intermittent and transient faults and is strongly affected by the accuracy of the judgements coming from error processing. On the other hand, the error processing mechanism performance heavily depends on the continued activity of failing components, as well as on the unnecessary removal of healthy components because of too conservative fault diagnosis. An integrated approach to the design of the error processing strategy and that of the α-count mechanism is therefore highly advisable: for any given set of application dependability requirements the error processing configuration and the α-count parameter settings can not be chosen independently each other to attain the best overall performance.

The α-count-based fault-treatment can be applied in several parts of the GUARDS architecture.

At the inter-channel level, each channel is endowed with an independent α-count, monitoring all the channels; the sources of diagnostic information can be the channel adjudicators and the output consolidation network. The former may each feed the information to their channel α-count using standard communication mechanisms; the latter should be provided with an auxiliary hardware output, to be fed back into every channel input (several implementation issues arise here, open to discussion). The inter-channel fault passivation mechanism should take the final decision upon removal of a channel through some form of voting on the signals coming from individual α-counts.

At the intra-channel level, an α-count is meaningful only if replaceable modules can be singled out from the channel structure, e.g. processor or I/O boards. In such cases, each channel can have an α-count taking care of the internal channel module's behaviour; no hardware modification is necessary for the mechanism implementation. Some form of passivation of modules internal to a channel is to be specified.

Last, the interactive consistency network may be subject to α-count monitoring: upon an interactive consistency session, a misbehaving Network Element [Powell 1996] can be identified looking up the vectors exchanged in the protocol rounds; this information can then be fed from each NE to an ad-hoc α-count acting in its channel (the other non-faulty channels have of course the very same information). The format of this disagreement signal and its path from NE to the rest of the channel are to be specified.

Properly assigning values to its parameters, the α-count-based fault-treatment strategy can be suitably adapted to the specific needs of the different instances of the GUARDS architecture.

Although the original formulation [Bondavalli et al. 1995 a] of this fault treatment strategy explicitly addresses hardware faults, it could be easily adapted to faults in software components without additional complication. In this last case, what needs to be investigated is the procedure to apply when a software component is diagnosed as permanently faulty. Following the fault classification in [Powell 96], software components can only suffer from design faults, discriminated in permanent (or systematic) design faults and temporary design faults. As stated there, residual faults in well tested components should be mainly of the temporary kind, and software is not subject to physical degradation as it is for hardware. These considerations would suggest that less attention would be necessary to the consequences of permanently faulty software components; however, since perfectly tested programs cannot be assumed in the GUARDS context (thus implying the possibility of systematic faults) it becomes necessary/attractive to apply fault treatment to software components too.

It is well understood that a hardware component (processor), once diagnosed as permanently faulty, should be removed, repaired or replaced. A software component diagnosed to be permanently faulty could be removed and possibly replaced as well, but the costs involved in producing a new software or extensively debug the faulty one in order to fix the defects could constitute an obstacle to the employment of such a practice. Thus, other possibilities should be considered, possibly in relation with the specific kind of software at hand. For example, in control systems applications, faulty software would produce bursts of errors when the input trajectory intersects a "failure region" in its input space [Bishop 1993, Bondavalli et al. 1995 b]; during the period that the program is in this region failure, it will produce a failure, but, hopefully, by exiting this region the program would restart to produce correct results, provided it be reinitialized to a correct state. In this case, the α-count-based fault-treatment could be used to identify the software component crossing a too wide failure region (or too many small but close failure regions) and the consequent action to perform could be to stop for "a while" the execution of this component; after that, it can be put again in operation, opportunely reinitialized. A similar idea is applied in the FTP/AP architecture proposed in [Lala and Alger 88] to attempt the recovery of a failed version. There, the failed version is initialized to a cold start state and allowed to bring itself to a congruent state (with other versions) over time by open loop operation. Its output is masked, but compared with the voted output; then the version is restored if its output agrees with voted output for several iterations.

Moreover, appropriate refinements could be thought to improve the performance of this fault treatment strategy. For example the introduction of a double threshold, where exceeding the first threshold is interpreted as a hint that the component could be affected by a permanent fault, but still kept in operation with reduced trustworthiness in the results produced by it, while the second threshold remains the limit to be exceeded by the component to be considered permanently faulty.

CONCLUSIONS

This report proposes a variety of fault tolerance mechanisms to be used in instances of the GUARDS general architecture to support applications with dependability requirements. Both error processing and fault treatment approaches are identified. To cover significant classes of applications characterized by specific dependability requirements, a set of fault tolerant structures are offered and their usage in the GUARDS context discussed. The behaviour of the proposed approaches is described through a C-like language and the main support mechanisms required for a real implementation are enumerated. Being the redundancy for dependability purposes accomodated both at inter-channel and at intra-channel level in the GUARDS general architecture, solutions to fault tolerance at these two levels are presented separately, given the diverse implications associated with the employment of redundancy at these two levels.

The description style of this document is intentionally kept at a quite general level; this mainly because i) more discussions are needed and several issues of the architecture itself must be clarified, ii) no final decision has been taken so far about the supporting kernel interface. Moreover, a general agreement on the interest in the proposed fault tolerance structures is necessary before further detailing be worthy of.

Although in a preliminary and incomplete form, this report helps to understand the open issues of the general GUARDS architecture which still lack a solution, to identify additional mechanisms not explicitly foreseen in the current definition of the general framework, to guide the choice of GUARDS COTS components.

REFERENCES

[Agrawal 88] P. Agrawal, "Fault Tolerance in Multiprocessor Systems without Dedicated Redundancy," IEEE Transactions on Computers, Vol. 37, pp. 358-362, 1988.

[Anderson and R. Kerr 1976] T. Anderson and R. Kerr, "Recovery blocks in action: A system supporting high reliability," in *Proc. 2nd Int. Conf. on Soft. Eng.*, San Francisco, California, 1976, pp. 447-457.

[Arlat 1996] J. Arlat *Preliminary Definition of the Validation Policy*, ESPRIT Project 20716 GUARDS Report D3A1, LAAS-CNRS, 1996.

[Avizienis and Chen 1977] A. Avizienis and L. Chen "On the Implementation of N-Version Programming for Software Fault Tolerance During Execution," in *Proc. Int. Conf. Comput. Soft. and Applic.*, pp. 149-155, New York, 1977.

[Bishop 1993] P. G. Bishop, "The Variation of Software Survival Time for Different Operational Input Profiles (or why you Can Wait a long Time for a big Bug to Fail)," in *Proc. 23th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-23)*, Toulouse, France, 1993, pp. 98-107.

[Bishop 1995] P.G. Bishop, "Software Fault Tolerance by Design Diversity," in M. R. Lyu ed. "Software Fault Tolerance", John Wiley & Sons, New York, 1995, pp. 211-229.

[Bondavalli et al. 1995 a] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico and L. Strigini, "Rational Design of Multiple Redundant Systems: Adjudication and Fault Treatment," in Predictably Dependable Computing Systems, B. Randell J.-C. Laprie H. Kopetz B. Littlewood (Eds.), pp. 141-154, Springer Verlag , 1995.

[Bondavalli et al. 1995 b] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico and L. Strigini, "Dependability Models for Iterative Software Considering Correlation among Successive Inputs," in *Proc. IEEE Int. Computer Performance and Dependability Symposium (IPDS'95)*, Erlangen, Germany, 1995, pp. 13-21.

[Di Giandomenico and Strigini 1990] F. Di Giandomenico and L. Strigini, "Adjudicators for Diverse Redundant-Components," in *Proc. 9th Symposium on Reliable Distributed Systems*, Huntsville, AL, IEEE Computer Society, pp. 114-123, 1990.

[Harper and Lala 1990] R. E. Haper and J. H. Lala, "Fault-Tolerant Parallel Processor", Journal of Guidance, Control and Dynamics, 14 (3), pp. 554-63, May-June 1991.

[Kantz and Koza 1995] H. Kantz, C. Koza, "The ELEKTRA railway signalling-system: field experience with an actively replicated system with diversity," in *Proc. 25th Int. Symp. on Fault Tolerant Computing (FTCS-25)*, pp. 453-458, California, IEEE Computer Society, 1995.

[Knight and Leveson] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," IEEE Transactions on Software Engineering, SE-12 (1), pp. 96-109, 1986.

[Lala and Alger 88] J. H. Lala and L. S. Alger "Hardware and Software Fault Tolerance: A Unified Architectural Approach" in *Proc. 18th Int. Symp. on Fault Tolerant Computing (FTCS-18)*, IEEE Computer Society, pp. 240-245, Tokyo, Japan, June 27-30 1988.

[Laprie et al. 1987] J.-C. Laprie, J. Arlat, C. Beounes, K. Kanoun and C. Hourtolle "Hardware and Software Fault Tolerance: Definition and Analysis of Architectural Solutions," in *Proc. 17th Int. Symp. on Fault Tolerant Computing (FTCS-17)*, pp. 116-121, Pittsburgh, PA, IEEE Computer Society, 1987.

[Laprie 1995] J.-C. Laprie "Dependability: its Attributes, Impairments and Means," in Predictably Dependable Computing Systems, B. Randell J.-C. Laprie H. Kopetz B. Littlewood (Eds.), pp. 3-24, Springer Verlag , 1995.

[Lee t. al. 1980] P.A. Lee, N. Ghani and K. Heron, "A recovery cache for the PDP-11," *IEEE Trans. Comput.*, vol. C-29, no. 6, pp. 546-549, June 1980.

[Lorczak et al. 1989] P.R. Lorczak, A.K. Caglayan and D.E. Eckhardt "A Theoretical Investigation of Generalized Voters for Redundant Systems," in *Proc. 19th Int. Symp. on Fault Tolerant Computing (FTCS-19)*, pp. 444-451, Chicago, IL, USA, IEEE Computer Society, 1989.

[Mongardi 1993] G. Mongardi "Dependable Computing for Railway Control Systems", in Dependable Computing for Critical Applications 3 (C. E. Landwehr, B. Randell and L. Simoncini, Eds) Dependable Computing and Fault-Tolerant Systems, 8, pp. 255-277, Springer Verlag, Vienna, Austria, 1993 (Proc. IFIP 10.4 Work. Conf. held in Mondello, Italy, September 14-16,1992).

[Powell 1996] D. Powell. *Preliminary Definition of the GUARDS Architecture*, ESPRIT Project 20716 GUARDS Report D1A1 AO 5000, LAAS-CNRS, 1996.

[Randell 1975] B. Randell "System Structure for Software Fault Tolerance," IEEE Transactions on Software Engineering, SE-1 (2), pp. 220-232, 1975.

[Rennels 1984] D. Rennels "Fault Tolerant Computing: Concepts and Examples", IEEE Transactions on Computer, C-33 (12), pp. 1116-29, 1984.

[Siewiorek and Swarz 1982] D.P. Siewiorek and R.S. Swarz *The Theory and Practice of Reliable Systems Design*, Digital Press, 1982.

[Xu et al. 1995] J. Xu, A. Bondavalli and F. Di Giandomenico "Dynamic Adjustment of Dependability and Efficiency in Fault Tolerant Software," in Predictably Dependable Computing Systems, B. Randell J.-C. Laprie H. Kopetz B. Littlewood (Eds.), pp. 155-172, Springer Verlag , 1995.

[Wellings 1996] A. Wellings *GUARDS Architecture: Timing Analysis Considerations* ESPRIT Project 20716 GUARDS Report D1A4/TN/7006/A, University of York, 1996.