Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE
DELLA INFORMAZIONE

PISA

Adjudicators For Diverse-Redundant Software: Problem Statement

F. Di Giandomenico, L. Strigini

Nota interna B4-43
Agosto 1989

ADJUDICATORS FOR DIVERSE-REDUNDANT SOFTWARE

F. Di Giandomenico, L. Strigini
IEI - CNR, Pisa, Italy

Abstract

The adjudication problem arises when fault-tolerant components are realized using replication. It consists in the definition of a mechanism for the choice of a correct value as the output value for the redundant component, starting from the individual results produced by the replicas of which the redundant component is composed.
The purpose of this report is to investigate the use of probabilistic knowledge about errors/faults among the components of the system to obtain good adjudication functions.
First, a terminology for the specification of robust redundant components is introduced. Then, an *optimal adjudication function* is defined, which has the highest probability of producing correct results, given the information available to it. Last, a method for the evaluation of different adjudication functions, useful for the choice of the best adjudication function for a real application, is outlined.

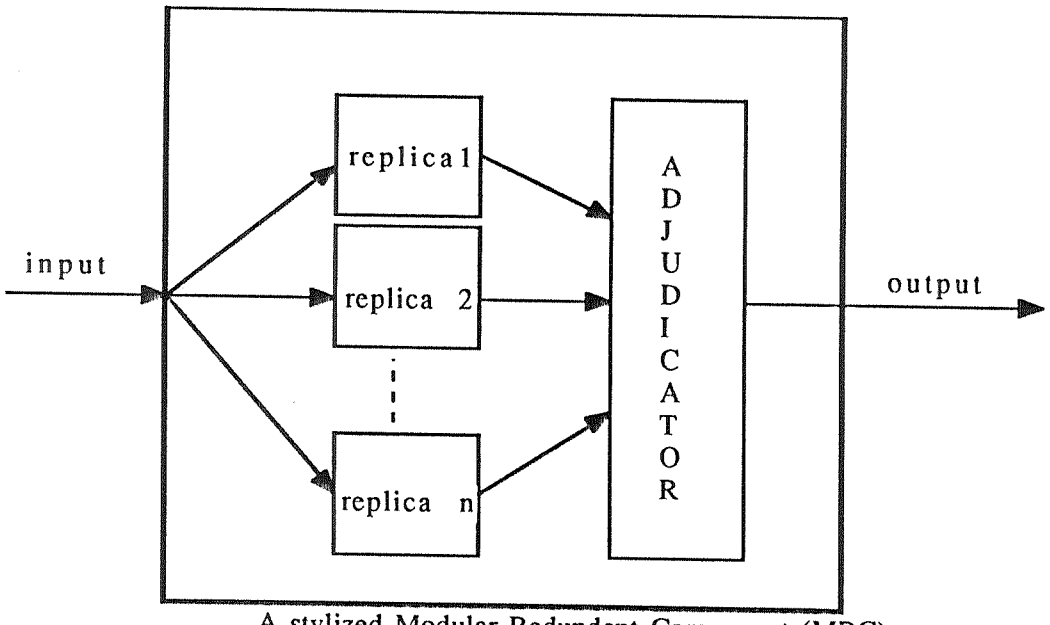
CONTENTS

1. INTRODUCTION

1.1. Informal statement of the problem

A technique to make a system capable of tolerating faults in hardware and/or software components is to use replicated hardware and/or software components in the system. A replicated component is substituted in lieu of an ordinary component, and consists of:
1) a set of subcomponents (that we call "replicas"), each one implementing the same function as the whole component, plus
2) some mechanism that obtains an agreed single result from the set of results produced by the replicas.
In particular, when design faults (either in hardware or software) must be tolerated, replicated components are often realized using *design-diversity*: the replicas, although built to the same specification, have different implementations in the hope that the design faults in different replicas will have error behaviours different enough to permit effective error detection and recovery or masking.

A redundant component can be represented as a set of replicas plus the mechanism that select the result for the redundant component from the results of the replicas. It is convenient to depict this mechanism as an additional component called *adjudicator* [Anderson 86]. In general, the adjudicator will receive a subset of the outputs due from the replicas, and from these it must either determine a single result or, if this is not possible, signal an exception. The result of the adjudicator becomes the result of the redundant component.



A stylized Modular Redundant Component (MRC).

Figure 1

The problem we consider consists in the definition of an adjudication function (that is, a function implemented in the adjudicator) that, under proper hypotheses on the system to which we refer, normally produces a correct result.

The obvious problem for this adjudication function in the selection of the correct value is that the results of some or all the replicas may be wrong (due to hardware faults and/or software bugs). Besides, the problem becomes more difficult when different correct values are allowed for different replicas with the same input. This generally happens when design-diversity is used.
We will call *syndrome* the information that the adjudication function can see (typically, the results of the individual replicas and possible diagnostic information) and use to derive its output value.

The adjudication problem is not new, but the adjudication functions normally used are generally quite simple, and try to select a correct value for the most usual syndromes, using simple algorithms.
The simplest adjudication function is the "exact" simple majority voting: the outputs of the replicas are compared for equality and, if more than half of these outputs agree, their value is selected as the output by the adjudicator. It has a limited application area (for example, it isn't usable with diverse software), so alternative algorithms have been investigated to cover the problem of adjudication, trying to obtain reliable adjudication functions for as many syndromes as possible. Examples are functions that realize "inexact" majority voting (that considers as equal two values if they differ no more than a quantity a-priori established), or generalized m-out-of-n voting, median voting, etc...
In general, simple adjudication functions can be proven always to give correct results if some fault combinations are assumed to be impossible. In describing systems that use such adjudication functions, the adjudication function is often left unspecified for these "impossible" syndromes. For example, the 2-out-of-3 voting function, works well so long as two replicas work correctly (and, indeed, the most probable syndromes in a system are those where all the three, or two out of three, replicas are correct). But the 2-out-of-3 voter is unable to give a correct output when three different values are observed in the system. As this syndrome really has a low probability, this function can be adopted in most applications.

In our studies, we want to take into account all fault combinations, because we are interested in problems of:
- ultra-high-reliability. The system failure probability is required to be so low that, for example, the use of 2-out-of-3 voting cannot be satisfactory, because when three different results are observed it doesn't guarantee a correct result, but this event happens with a probability that is relevant to obtain the required system reliability, and
- low-redundancy configurations. A low number of replicas are used to obtain a fault tolerant component, but we still want to get the best possible reliability from the redundant component: so, each possible configuration must be investigated to try to obtain a correct result as much as possible.

If arbitrary fault combinations are allowed, there is of course no adjudication function that can mask them all, and which adjudication functions are better than others depends on the probabilities associated to the individual fault combinations.

The aim of this report is to investigate the use of probabilistic knowledge about component failures in defining the adjudication function.

This report is divided in three parts.
Section 2 gives a precise definition of the terms we use, and defines an *optimal* adjudication function, based on the knowledge of the system failure probabilities.
Section 3 states assumptions about a simple systems we choose for a first study.
Section 4 deals with the problem of evaluating adjudication functions, including the definition of tools for the evaluation of different adjudication functions for realistic systems.

2. DEFINITIONS

2.1. Need of realistic definitions

According to a common definition, the specification of a program is seen as a function defined on the state of the storage of the computer and producing a new state of the storage. This definition has some drawbacks for the study of redundant components, viz.:

- the specification of a program often describes a set of acceptable outputs for every input value, rather than a single required output value. This must be considered, since distinct implementations of the same specification may yield different but correct results;
- the inputs explicitly considered in the specifications of a software component are usually limited to the state of a small subset of the storage of the computer: the required output is an invariant with respect to the contents of most of the storage space. In reality, the output can be influenced by the contents of the rest of the storage, but this appears to an observer as a non-deterministic behavior of the program;
- the programs are considered as strictly sequential. In reality, sequential application programs usually share a physical machine, through an operating system, and input/output activities proceed in parallel with the execution of every program;
- erroneous behavior often appears to be non-deterministic, for all the above reasons.

We shall try to define our terms in a way closer to their intuitive meaning in common usage.

2.2. Component specification

A software component is defined in terms of its external behavior, i.e., the values of the outputs it produces as a consequence of receiving inputs.
For our purpose, the life of a software component is seen as a series of one or more invocations. During each invocation the component runs and interacts with the rest of the world via inputs and outputs. We shall not consider the exact sequence of interactions, but lump together all the information received by a component during an invocation, and all the information emitted by it, into two items of information called the value of the input vector and the value of the output vector for that invocation.

2.2.1. Input and output

The *input vector* and the *output vector* of a component are each a set of mathematical variables, obtainable by observing the system hardware according to rules specific to each individual computer and software component. E.g., the values of the output variables of a procedure may be obtained by observing the state of the stack when control returns from the procedure to the calling program. Input and output variables need not be the contents of memory elements but can be arbitrary observable values, e.g. the response time of a procedure. Of course, we expect the specification of a component to define its input and output vectors in a way corresponding to the intuitive meaning of such words. An *input (output) value* is a value of the *input (output) vector*, i.e. a set of one value for each input (output) variable. In accordance with common usage, we shall often use the word "input(output)" instead of "input(output) value".

For a generic software component, we shall call I and O the universes of the possible inputs and possible outputs, respectively[1].

We wish to describe the behavior of the component by a deterministic function. Therefore, we describe the factors that may influence the behavior of the component, and are not functions of the input vector, by an *external influence vector*. The set of all the possible values of the external influence vector define the external influence space, E. The vector obtained by concatenating the input vector and the external influence vector will be called the *generalized input vector*. The concatenation of two vectors i and j will be denoted by (i conc j). The sets of all possible values for each of these three vectors define three Cartesian spaces, and the generalized input space is the Cartesian product of the input space, I, and the external influence space, E: $I^G = I \times E$.

2.2.2. Specification function of a component

The way a component must work is defined by a specification. This includes (among other requirements) a function (*specification function*) that maps elements of I into elements of the set of the subsets of O (which we shall call Ω):

$F: I \longrightarrow \Omega$.

For each $i \in I$, F(i) indicates a set of *correct* outputs, contained in O[2].

---

[1] Our definition of "inputs"

1) allows one to consider as input variables only those variables that it makes sense to mention in a specification to be delivered to a programmer; by contrast, [Cristian 87] includes the whole of system memory, i.e. physical variables that are not visible to an application programmer;
2) is not limited to memory contents at invocation time: it may include all kinds of information made available to the component from the outside, such as messages received during its execution, interaction with other, concurrently executing programs via shared memory, hardware events (e.g. interrupts). Of course, inputs to software components from other software components that share the same CPU always take the form of values written in memory (even timing information is obtained by reading information in memory).

[2] It is normal for the specification of a component to contain "don't care" values for some variables, depending on the values taken by other variables. E.g, when an error flag is 1 any value is acceptable for all variables except some that are specified as error codes, and when the error flag is 0 any value

It is considered good practice for F to be defined for every $i \in I$[1]. We shall normally assume this to be the case. The usual real case where there is a subset I^u of I where F is not explicitly defined by the specifiers can be described by saying that over I^u any output is permitted, i.e. defining F(i)=O for any i in I^u.

> It may be useful to consider the output vector as composed of several subvectors, and split the specification function into several distinct functions, each defining the desired values for one subvector. So, if a component simultaneously performs several tasks in a system, one can study its performance of each task independently.

2.2.3. Behavior of a component

The specification function defines the ideal behavior of a component; a real implementation of the component can deviate from its specified behavior, due to external influences (that can lead to *operational faults*) and/or to the implementation of the component (that can contain *design faults*).
Let us consider a component A, produced according to a specification with specification function F. We can distinguish two functions describing the behavior of the component A:

- the *nominal behavior function*: $F_A^N : I \dashrightarrow \Omega$ and
- the *physical behavior function*: $F_A^{Ph} : I^G \dashrightarrow O$.

The nominal behavior function represents the behavior of the component in the absence of operational faults, and gives, for each input value, the set of outputs possible for the program (it may describe a non-deterministic behavior)[2]
The physical behavior function describes the actual output of an implementation of the component for the generalized input value (i.e. the values of the input *and* the external influence vectors). So, the physical behavior of a component is determined by three factors: the program of the component itself (source code or some other chosen representation), its

---

is acceptable for the error codes. So, the sets of outputs acceptable for a given input value often have peculiar "shapes", e.g. hyperplanes in a Cartesian hyperspace.

[1] It seems reasonable to request that the specification includes the definition of the subset of legal inputs, and the outputs include error messages or exception signals.

[2] Non-determinism in reality may be due to two reasons:

a) a component may be an inherently non-deterministic program, due to
   a1) a design error, e.g. because the programmer mistakenly used the value of a variable before initializing it, or to
   a2) some other reason, e.g. dependence on some external event (message, interrupt) or non-deterministic function (random number generator);
b) an operational fault may cause a deterministic program to produce an unpredictable result (this does not influence the function F^N).

We shall call deterministic components those whose nominal behavior is deterministic, i.e. a component may be called deterministic even when affected by type b) non-determinism.

inputs, and all other factors, that we call external influences[1]. According to our definitions, erroneous behavior can be caused either by the program (due to a *design fault*) or by external influences (due to an *operational fault*). Since we assume the specification function to be defined over all I, **inputs can not cause errors unless design faults are present.** By definition, a component which delivers an $F^{Ph}(i) \notin F(i)$ is faulty, and $F^{Ph}(i)$ is an *erroneous* output.

The function F_A^N is related to F_A^{Ph} as follows:

$F_A^N(i)$ is the set of all the values of $F_A^{Ph}(l)$ for values of l of the form (i conc j) with j such that no operational fault exists:

$\forall i \in I$, $F_A^N(i) = \{F_A^{Ph}(l) \mid l = (i \text{ conc } j), j \in E_H\}$, where $E_H \subseteq E$ is the set of healty values of the external influences, obtained by excluding all the values which implys the presence of operational faults.

2.2.4. Additional properties of software components.

Among the requirements that are not part of a specification function, an example is *determinism.*

It may be useful to require that a component behave deterministically, i.e. that

$\forall i \in I \ |F^N(i)| = 1$

even if the specification function does not imply such a requirement, i.e.

$\exists i \in I$ such that $|F(i)| > 1$.

Requiring determinism implies that any individual component must always produce the same result for a given input, without, of course, implying any such similarities between two components built from the same specification. For a component that is required to behave deterministically, non-determinism of the nominal behavior implies a design fault.

Another important example is what we call *robustness*, which will be discussed in paragraph 2.3.

2.2.5 Termination and non-termination

An interesting issue is *termination*. A program whose execution does not terminate does not produce a result in any conventional sense[2].But the rest

---

[1] The boundaries between the three factors are somewhat arbitrary: we only request that they be defined precisely, for a given component. For instance, the influence of a compiler could be included in the "program", considering the latter in its machine-code representation, or in the "external influences" (considering the program to coincide with its source code). The value of an uninitialized memory area could be considered as an input or as an external influence. The value of a configuration parameter could be considered as part of the program or as part of the input. Operator errors that cause component failures could be considered either as input values that trigger design faults, or as values of external influence variables implying an operational fault, that causes the failure of the component. As for the tolerance of such an error, it would be labelled as the treatment of an input exception in the former case and as robustness of the component (which is treated later in this appear) in the alter.

[2] The problem of non-termination cannot be solved by defining the termination time as an output variable, since, if the component execution

of the system must be able to cope with the possibility of a non-terminating program.

So, the specification of the component must include an output variable that may be used by other components as a time-out signal: e.g., there might be a boolean output variable TIMEOUT defined as "its value is 1 if the queue of incoming messages from the component to some other component is empty when the value of a real-time clock reaches a given threshold". If such an output is defined and used as input to other components, the non-termination of a component becomes observable as an output of the component. Otherwise, we could assume the existence of an output value called NONTERMINATION, which is a possible output of the experiment "invocation of the component", but *is never observable by other components in the system.*

2.2.6 Specification of exceptional behavior

One normally tries to specify not only the ideal result that a component should give if all goes well, but also the behavior required if the component cannot deliver this ideal result.
We can distinguish here (along with [Anderson 85]) between "interface exception" and "local exception".
The former are said to occur when the component is invoked with illegal inputs. These exceptions are easily included in the specification function: the specification function maps illegal inputs into specific exceptional outputs.
All other exceptions are called "local exceptions": they are caused by, e. g., internal failures, or undesired conditions (overflows, refusal of service component).

Two typical ways for dealing internal exceptions are:
- for each input value, not only the normal desired result is specified, but also alternative, exceptional results. There is an implied preference for successful results, and the circumstances in which the exceptional results should be produced are not detailed;
- the input variables considered include a number of observable indicators of exceptional situations (e.g. an overflow flag), and the specification function indicates in detail which circumstances should produce which kind of exceptional results. A "catchall" clause is often included, viz "if it is clear that the ideal services can't be delivered, and none of the detailed exceptional condition in a list is verified, an "unidentified exception" output is required".

2.2.7. Subsets of O

The universe O of possible outputs can be divided into convenient subsets, defined together with the specification of the component. First, based on the specification function F, the subsets O^l of *legal outputs* (i.e, those output values that are allowed by the specification for some input value) and O^i of

---

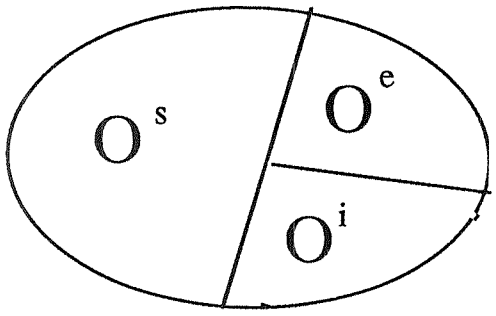
does not terminate, the value of this variable can never be observed. In general, our definition of output variables as results of physical observations always implies the possibility that the conditions specified for the observation procedure to start never get satisfied: e.g., if we call "outputs" of a procedure the values observed on the stack when the procedure returns, it may happen that the procedure never produces outputs.

illegal outputs (i.e, those output values that are not allowed by the specification for any input value) can be defined as

$O^l = \bigcup_{i \in I} F(i)$ and $O^i = O - O^l$.

Only faulty components can produce illegal outputs; of course, a faulty component may also produce legal outputs.

Typically, O^l is further divided into at least two subsets, to specify the relative desirability of different outputs, or equivalently, the degree to which an output satisfies the intents of the specifier. This minimum subsets of O^l could be called O^s and O^e, the *successful* results and *exceptional* results[1]. Their intuitive meaning is: if the component actually accomplished what it was requested to do, it should produce an output value belonging to O^s; if it could not do it (due to some exceptional situation taken into account at the specification definition), it should produce an output value belonging to O^e (and the rest of the system must take into account this failure[2]). **The actual worth of results in the operation of a system depends on how the component is employed in the system.**



A typical partition of O.

Figure 2

2.2.8 Subsets of I.

For the evaluation of how good a component is, it is interesting to define subsets of the input universe I.
Conventionally, ([Cristian 87]), if the software component is seen as a deterministic function of its input values, an input value belongs to one, and one only, out of four partitions, as follows.

1. the subset for which the specification function is not defined, I^u;
2. the subset on which the software component produces a wrong result, I^f;

---

[1] Of course, a (usually bad) specification may have an empty O^e.

[2] In a well-designed system, it is usually possible to designate, in the output, a bit of information that we shall call an *error flag*, which specifies whether the operation of the component was normal or "exceptional" (and takes the value 1 in this latter case). We shall sometimes consider this bit of information separately from the other information designated as output of the component.

3. the subset on which the software component produces a correct, successful output, I^s;
4. the subset on which the software component produces a correct, exceptional output, I^e.
By assessing the size of these subsets, we can assess the quality of the component.

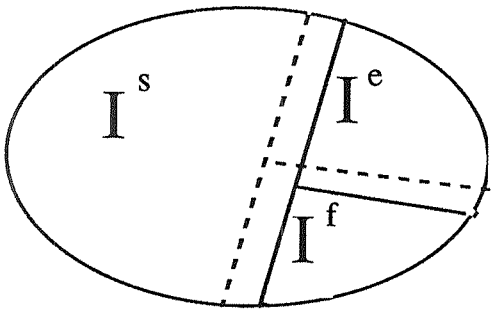
Two problems arise:
1) with our non-deterministic model of a software component, the definitions above no longer define a partition on I;
2) to determine the actual worth of a component in the system that includes it, the output must be classified in a way that depends on the system and may lead to different relevant partitionings of the input space (with fewer, more, or different subsets).

Problem 1) is easily solved if for our system the relevant classes of outputs are indeed: "correct, successful", "correct, exceptional", and "wrong". A convenient definition of the partition would be:
$I = I^s + I^e + I^f + I^u$, where
- I^s, the *successful domain*, is the set of all i such that $F^N(i)$ consists of correct, successful outputs:
  $I^s = \{i \mid i \in I, F^N(i) \subseteq (F(i) \cap O^s)\}$;
- I^e, the *exceptional domain*, is the set of all i such that $F^N(i)$ consists of correct outputs, including **at least one exceptional output**:
  $I^e = \{i \mid i \in I, F^N(i) \subseteq F(i), (F^N(i) \cap O^e) \neq \varnothing\}$;
- I^f, the *failure domain*, is the set of all i such that $F^N(i)$ **either** includes **at least one erroneous output, or** violates **some other requirement** of the specification (e.g., determinism):
  $I^f = \{i \mid i \in I \mid ((F^N(i) \not\subset F(i))$ **or** (some other requirement is violated))$\}$
- I^u, the *undefined domain*, is the set of all i for which F(i) is undefined:
  $I^s = \{i \mid i \in I, F(i) = O\}$.

2.2.9 Subsets of I^G.

To take into account all kinds of faults, we can also define subsets of I^G with respect to the physical behavior of the software component. This is deterministic (as it is the actual behavior, taking into account all external influences), and he same four-way partition can be applied as defined over the input set in [Cristian 87].

ordering on all the implementations of a specification. The Figure 4 shows this with reference to an example of 3-subset partition of I.



The partition marked with the dotted line corresponds to an implementation less robust than the one marked with the solid line.

Figure 4

2.3.2. Statistical robustness

Statistical robustness gives a total ordering among components built to the same specification.

To state statistical robustness properties of a component, one must first specify an experiment, i.e. a given distribution of generalized input values.

If I^G is considered partioned in only two subsets (e.g.: inputs for which correct results are produced and those for which wrong results are produced), then statistical robustness has an intuitive meaning: A is more robust than B iff the probability mass associated with the "better" input subset is greater for A than for B.

If more subsets are of interest, as in our example above, then several orderings become possible.

In our example, we could define two orderings, that is A can be called statistically more robust than B if:

1) the probability that an input i belongs to the set of input values I^{Gf}, for which an erroneous output is produced, is greater for B than for A, that is: $P(i \in I^{Gf}_A) < P(i \in I^{Gf}_B)$; this could be called statistical safety; or

2) the probability that an input i belongs to the set of input values I^{Gs}, for which a correct, successful output is produced, is greater for A than for B, that is:

$P(i \in I^{Gs}_A) > P(i \in I^{Gs}_B)$.

Statistical robustness is normally measured experimentally, with the distribution of generalized input values defined as part of the experimental set-up. In theory, if one knew the F^{Ph} of a component, one could define the distribution over I^G, and derive the probabilities above analytically.

The experimental hypotheses must include (implicitly or explicitly) the probabilities of the different kinds of operational faults. It is normal to restrict these by simple assumptions, even with extreme hypotheses as the absence of all operational faults.

With our definitions, if A is logically more robust than B it is automatically more robust statistically (provided the input set considered in the experiment is a subset of that used in evaluating logical robustness). This establishes the importance of the concept of logical robustness: if A can be proven to be logically more robust than B, it is proven to be more robust statistically as well, for all experiments over the same input set.[1]
To obtain global "statistical robustness" ordering among implementations of a component, one must assign "worths" to output subsets, rather than just an ordering. Then, the worth of a component is the expected value of the worth of its output for the experiment of interest.

2.3.3. Generalized specification functions

Let us give an example of the set of subclasses of outputs, that may be of interest for an evaluation. Assume that, for a given input, both a successful and an exceptional result are allowed. A possible ordering, by worth, of the subsets of O is:
1- a correct, successful result;
2- a correct, exceptional result;
3- a wrong, exceptional result;
4- a wrong, successful result.
One could further refine this subdivision by observing that some exceptional outputs are better than others, and some successful outputs better than others as well, and some wrong outputs are worse than others.
An extreme point of view is that, knowing the application of a component, its specification function should be defined as mapping each input value into a fuzzy set of "desired outputs": an output can be desirable in degrees varying from 0 (totally undesirable) to 1 (the best result for that input). Equivalently, the specification function could map each input point into an objective function over the space O, indicating the value for the enterprise of each possible output for that input. Statistical robustness would be given by the expected value of these functions for the experiment given.

> Such descriptions of what is desired from a software component, though appealing for systems with complicated interactions with their environment, seem to have no practical value for "small" software modules (such as procedures and functions), because: 1) such descriptions are not usually available in practice; 2) sensible software development practice often aims at determinism, and the behavior expected for a given input is usually limited to some small set of values (under no or limited faults) or total unpredictability (under more general fault hypotheses): **knowing the relative undesirabilities for the user of all possible errors is of little use to the developer**; 3) for small components, the "worth" of the output would change with the use of the component: e.g., for a procedure, the specification would depend on the point from where the procedure is called; this is **no use to a developer that must** produce standard re-usable components.

2.4. Modular-redundant component

Given a specification S for a software component, let us separate the output vector into a *functional output* part and all the other output variables

---

[1] Of course, statistical robustness relationships among components of two systems have no **direct** bearing on the overall robustness of the two systems.

(typically, diagnostic outputs). A component C built to specification S is called a *modular redundant component (MRC)* if it contains as subcomponents more than one SCs (*replicas*), such that:
- all replicas are built to a same specification S';
- S', limited to the functional outputs, has the same specification function as S.

In particular, the replicas have the same input and output vectors as the MRC, limited to the functional parts of these vectors. The specification of the MRC may differ from that of the replicas in the diagnostic outputs, the performance requirements, and such; besides, the MRC is usually required to be more robust (in some sense) than each of the individual replicas.

If two or more replicas are copies of a same software component they are said to belong to the same *variant*.

2.5. Stylized modular-redundant component

We define a *stylized MRC* as depicted in Figure 1:
- there is one *adjudicator* component, receiving the outputs of all the replicas;
- the inputs to the MRC are used as inputs by all the replicas; i.e., the underlying system guarantees that all the replicas normally receive identical inputs.

The outputs of all the replicas are fed as inputs to the adjudicator. The output of the adjudicator becomes the output of the MRC.

2.6. Adjudication function

The adjudication function is the function implemented in the adjudication mechanisms (or, the specification function of the adjudicator module when considering a stylized MRC as in Figure 1).

This function is defined on the input universe constituted by the output values of the replicas.
The output of the adjudication function is a single value that becomes the output of the MRC. It is either an agreed successful value or an exceptional value when no agreed value is found.

The purpose of the adjudication function is to select a correct output for the MRC, i.e., a value that satisfies the specification function of the MRC.

Let F: I----> Ω, be the specification function for the MRC.
To avoid confusion, we will call *syndrome* the argument of the adjudication function. Let SYN be the input universe for the adjudication function. An element of SYN is a tuple composed of the results of all replicas for a given input to the MRC.
For each input i $\in$ I to the MRC, a tuple exists in SYN composed of the results of each replica for the input i. Let $T_F(i)$ be such tuple of n elements (n is the number of replicas in the MRC) relative to the input i, where the k-element $T_F(i)[k]$ is the output produced by the k-th replica of the MRC on the input i.
Then, a specification for the adjudication function could be:
the adjudication function for a given syndrome s must produce an output that, according to the specification function of the MRC, is produced by the MRC for that input i for which the output of the replicas constitute the syndrome s to the adjudication function, that is a function

F_{adj}: SYN ---> O such that,
$\forall s \in SYN$, if $(F_{adj}(s)=o)$ then $(\exists i \in I \mid (o \in F(i)$ and $T_F(i) = s))$

We will call an adjudication function that behaves according to this specification the *idealized adjudication function.*

> The degree to which a specified adjudication function meets its idealized behavior depends, among other things, on the additional information available to it, beside the outputs of the replicas.

In general, given the specification function F for the MRC and an input i, the output produced by the implemented adjudication function can be:

- a value o such that $o \in F(i)$, that is a *correct* value (successful or exceptional);
- a value o such that $o \notin F(i)$, that is a *wrong* value, successful or exceptional.

Clearly, a correct value is "better" then a wrong value, and a wrong, exceptional value is "better" than a wrong successful value.

2.7. The optimal adjudication function.

As said in the Introduction, the problem of defining an adjudication function has been addressed in the literature, and a set of solutions have been investigated. They consist of generally simple adjudication functions that work well under limiting hypotheses about failures in the system.
In reality, some applications (such as critical applications) may require that every fault combination be taken into account, because their effects are relevant to obtain the required reliability.
There is no adjudication function that can mask all conceivable fault combinations. This is easily proven by considering that different possible fault configurations can lead to the same observed syndrome. Which output value is correct may depend on which of these possible fault configurations is the actual one. Then, the real fault configuration can only be guessed with a non-null probability of being wrong, and the accuracy of the guess depends on the failure probabilities for the system components.

With this knowledge, one can produce an *optimal* adjudication function, i.e., one that has the highest theoretically possible probability of producing a correct result for any input to a particular MRC.
An adjudication function that for any syndrome chooses that result that has the highest probability of being correct, conditional on the occurrence of that syndrome, is such an optimal adjudication function.
In fact, the probability that an adjudication function F_{adj-i} produces a correct result is given by

$$(1)\quad P(F_{adj-i} \text{ produces a correct output}) = \sum_{s \in SYN} P(F_{adj-i} \text{ produces a correct output} \mid s) * P(s),$$

where s is the event: syndrome s is produced.

That adjudication function F_{adj-*} that, for each syndrome s, chooses that output o* such that:

$$P(o^* \text{ is correct} \mid s) = \max_{o \in O} \{P(o \text{ is correct} \mid s)\}$$

maximizes expression (1), and hence is the function that has the highest probability to select the correct value, that is the optimal adjudication function.

Knowing the conditional probabilities needed is of course a hard problem.

3. A SIMPLE ADJUDICATION PROBLEM

We describe here a simple practical adjudication problem taken from the Delta-4 distributed system [Powell 88].

3.1. Hypotheses on the system.

1. The MRCs in the system are stylized MRCs.[1]

3.2. Hypotheses on the replicas

3.2.1. Determinism

Unlimited non-determinism complicates the treatment of the adjudication problem. Therefore, we impose the following constraints.

All the parts and transitions of the machine status that may affect the behavior of the replicas in the absence of (design or physical) faults are considered as inputs, and, as stated above, the system is so built as to provide identical inputs to all the replicas. (Hence, once the inputs are defined, any program that is non-deterministic contains, by definition, a design fault). This requirement is satisfied, for instance:

- with respect to variables used before initialization, if such use is considered as a design fault, or made impossible by the compiler, or if their values are preset to a default value when the program is loaded, or considered as part of the input;
- with respect to the ordering of messages sent to a replicated software component, if the system guarantees that they are delivered in the same order to all replicas, or if all variants are written in such a way that their outputs do not depend on the order of reception of the messages.

As a result of these assumptions, we have that, in the absence of physical faults, all the replicas of a same variant compute the same output value (i.e., the value of the nominal behavior function of the variant), except for inputs belonging to the failure input domain of the variant. The actual outputs of the replicas depend on their nominal behaviors and the physical faults present, and possibly, if the replicas contain design faults, on other factors not considered as inputs.

---

[1] In the Delta-4 architecture, the adjudicator software component, and the input demultiplexing software, i.e. the hard-core of such an MRC organization, run in a particular hardware component (fail-silent NAC), to make the scheme viable. In all MRC schemes with an identifiable adjudication module, the adjudicator is assumed to run on a separate, protected hardware component (or to be replicated on several hardware components), so that it can be considered fault-free.

Assuming total determinism by all the replicas in the MRC is a strong constraint to implement in practice in current systems. On the other hand, no hypothesis is made on how faulty replicas behave: for instance, a software fault may produce different results by two replicas of the same variant.

3.2.2. Self-checking replicas

Each replica in the MRC performs an acceptance test on its own results; the outputs of each replica include a boolean variable *errorflag* (equal to 1 if the other outputs are judged to be correct by the acceptance test, and 0 otherwise).

3.2.3. Non-triviality

We impose a *non-triviality* constraint on the specification:
$\exists$ i,i'$\in$ I such that $F(i) \cap F(i') = \varnothing$,
and on the implementation:
$\exists$ i,i' $\in$ I such that $F_k^N(i) \cap F_k^N(i') = \varnothing$.
In other words, programs that [are required to] give the same result for all inputs are excluded.

3.3. Information available to the adjudicator.

For an invocation of the MRC, the adjudicator knows:
- the array of the values of the output vectors of all the replicas;
- for each replica, the information of which node it runs on and which variant it belongs to.

Besides, the only firm hypothesis on the results of two replicas is:
- the results of two replicas belonging to the same software variant, if correct, are identical.

As a further constraint, the Delta-4 scenario requires that the adjudicator:
a) produce as a result one of the outputs of the replicas;
b) use *only* the set of self-check results by each replica and the set of the results of pairwise comparisons among the output values of all the replicas.
Both are sets of boolean values, where "0" means "disagreement" and "1" means "agreement". The resulting N-element vector and N×N matrix constitute the *syndrome* of our system.

4. EVALUATION OF ADJUDICATION FUNCTIONS.

4.1 A method for a realistic evaluation of adjudication functions.

To choose among different candidate adjudication functions for a system, it is necessary to have a realistic evaluation criterion. A first evaluation for a given system configuration, can be derived observing their behaviour on a number of experiments. Given a syndrome, each possible adjudication function will produce a value that can be correct or erroneous. In general, there is no guarantee that we shall find an adjudicator that is consistently the best for every syndrome. In fact, without any further knowledge, it is impossible to choose between two adjudicators if one behaves better for a syndrome and the other for another syndrome.

Let an *error state* be defined as a set of variables, describing the system during an invocation of the MRC, that allow the deduction of:
- the correct result for the MRC,
- the syndrome observed by the adjudication mechanism in the MRC.

For each adjudication function F_{adj-k}, the probability that it produces a correct result is:

(2) $P(F_{adj-k} \text{ produce a correct result}) = \sum_{x \in STATES} P(x)*U_k(x)$,

where:
- STATES is the set of possible error states;
- P(x) is the probability of occurrence of the error state x;
- $U_k(x)$ is a binary function whose value is "1" if the value produced by the function F_{adj-k} is a correct value for the error state x, "0" otherwise.

Error states may exist with different correct results but the same syndrome. The correctness of the result given by the adjudication function for a syndrome depends on which is the actual error state that produced that syndrome.

The same probability expressed by (2) can be written as:

(3) $P(F_{adj-k} \text{ produces a correct result}) = \sum_{s \in SYN} P(F_{adj-k} \text{ produces a correct result} \cap s)$,

where s is the event "the value of the syndrome is s", and:

(4) $P(F_{adj-k} \text{ is correct} \cap s) = \sum_{x \in STATES} P(s \mid x)*P(x)*V_k(x)$,

where :
- $P(s \mid x)$ is the probability of observing the syndrome s conditional to the occurrence of the error state x. Since an error state deterministically implies a syndrome, $P(s \mid x)$ is "1" if the error state x implies the syndrome s, "0" otherwise.
- $V_k(x)$ is a binary function having value "1" if the output of F_{adj-k} for the syndrome implied by the error state x is the correct output for the error state x, "0" otherwise.

Supposing we know the system configuration and information about probabilities of relevant events in the system, the steps to derive the comparison are:

1- enumerating all the possible error states x for the given system configuration, and assigning their probabilities;
2- enumerating all the possible syndromes in the system and assigning to each such enumerated syndrome s the conditional probability $P(s \mid x)$, which can be "0" or "1", for each error state x of step 1;
3- for each syndrome of step 2, evaluating each adjudication function according to expression (4).

Then, the total probability that an adjudication function produces a correct value, with the given system configuration and assigned probabilities, is found computing expression (3).

4.2 The probability assignment problem.

The main problem in this evaluation is the difficulty of finding the correct probabilities of all the error states (component failures, faults of components that affect other components, etc..). In general, only estimates of these probabilities for some system components are known.

In a typical real-world system of the kind we described in chapter 3, we can assume we know:
- hardware fault rates for the nodes of the system;
- some idea of the joint probabilities of hardware faults in different nodes (or the knowledge that they are independent);
- software error rates for each variant;
- "something" about joint error probabilities for different software variants;
- "something" about error probabilities for the self-checks of the replicas, and the probabilities of joint errors with the replicas themselves;
- "something" about the conditional probabilities of observing a syndrome s when the error state x occurs.

This is not enough: for example, the probability that a hardware fault will produce an error in the software components running on it is generally unknown.
To obtain some kind of assessment, we can try and assign 'invented" probability values to those events for which nothing can be hypothesized from initial estimates.
The probability values so obtained constitute an approximation of the real probability distribution of faults and errors, and the precision of this approximation depends on how much information is known on the component failures (and on how trustworthy it is), and on the algorithm chosen to invent the unknown probabilities.

4.3 How to choose the best adjudication function.

We have defined the optimal adjudication function; but this is not always the best adjudication function to use in a MRC. For example, performance requirements could make it unusable (it requires more time than simpler adjudication functions), or, for a given system configuration and assigned probabilities, the reliability of the optimal adjudication function could be comparable to that of another function that is better under other points of view (for example, it follows a simpler algorithm).
The evaluation method described in the previous paragraph can help in choosing the best adjudicator, given a real system configuration.
The choice of the best adjudication function is very sensitive to the probability assignment. The probability that a result r produced by the adjudication function k for a given syndrome s is wrong grows with the probabilities of the error states from which the syndrome s can be derived and for which the result r is wrong.

The steps to follow for choosing a good adjudication function to use in a given system are:
1) assuming some initial probabilities. They depend on the particular system under examination (for example, low redundancy systems or ultra-high reliability systems), and are values that are plausible in these systems or are concerned with particular situations believed to be interesting (for example, totally symmetric or asymmetric system);

2) inventing probabilities for each error state for which nothing is initially known. These values must be assigned in such a way that:
   a) they are consistent with the initial probabilities and the axioms of probability theory;
   b) they are plausible for the system under examination;
3) evaluating different adjudication functions for this probability assignment, supposed to be fairly realistic;
4) giving different values to probabilities at point 2), according to some significant criterion trying to guess the real values for these probabilities, and returning to point 3);
5) giving different values to probabilities at point 1), as they are estimates of the real values and perhaps different possible values are plausible. Then, iterate the steps until an adjudication function, that behaves in a good way for the different probabilities assignments examined, is found. This adjudication function is the best one, as it shows the best behavior for the probabilities assignments supposed to be good realistic approximations.

Neverthless, a difference could be observed between the supposed behavior of an implementation of an MRC and its actual behavior, also when the adjudication function is chosen using this evaluation method. It depends on the error made in the assignment of the required probabilities.

5. CONCLUSIONS

In this report, the problem of adjudication for redundant components has been addressed. The main results consist in the definition of:
- a terminology for the problem of specifying robust systems;
- an optimal adjudication function, which gives the theoretical upper bound for the goodness of all adjudication functions;
- a method for a realistic evaluation of different adjudication functions.

Both the optimal adjudication function and the evaluation method utilize information about probabilities of occurrence of the most relevant events in the system. Realistically, only an estimate for some of these probabilities can be assumed as known.
To have good results in real applications from this optimal adjudication function and evaluation algorithm, methods for computing accurate values for the required probabilities should be investigated (as suggested in paragraph 4.3).

Our future work will consist in the construction of a tool implementing the evaluation method described in this report. It will be useful for having real comparison values of different adjudication functions. For the problem of probabilities assignment, we will investigate solutions utilizing already existing applications in mathematical field.

6. REFERENCES

[Anderson 85] T. Anderson, ed., "Risilient Computing systems", 1985
[Anderson 86] T. Anderson, "A structured decision mechanism for diverse software", fifth Symposium on Reliability in Distributed Software and Database Systems, Jan 13-15 1986, Los Angeles, California.
[Arlat 87] J.Arlat, K.Kanoun,J.C.Laprie "Dependability evaluation of software fault-tolerance", LAAS Research Report n. 87.389, December 1987

[Cristian 87] F. Cristian, " Exception handling", IBM Research Report RJ 5724, 1987.

[Echtle 89] K. Echtle, "Distance Agreement Protocols", nineteenth International Symposium on Fault Tolerant Computing, June 1989, Chicago.

[Echtle 89] K. Echtle, "Fault diagnosis by combination of absolute and relative tests", presentation in Toulouse, March 1989.

[Lorczak 89] P.R. Lorczak, A.K. Caglayan and D.E. Eckhardt, "A theoretical investigation of generalized voters for redundant systems", nineteenth International Symposium on Fault Tolerant Computing, June 1989, Chicago.

[Powell 88] D. Powell, ed , "DELTA 4 Overall System Specification", Issue 2, November 1988