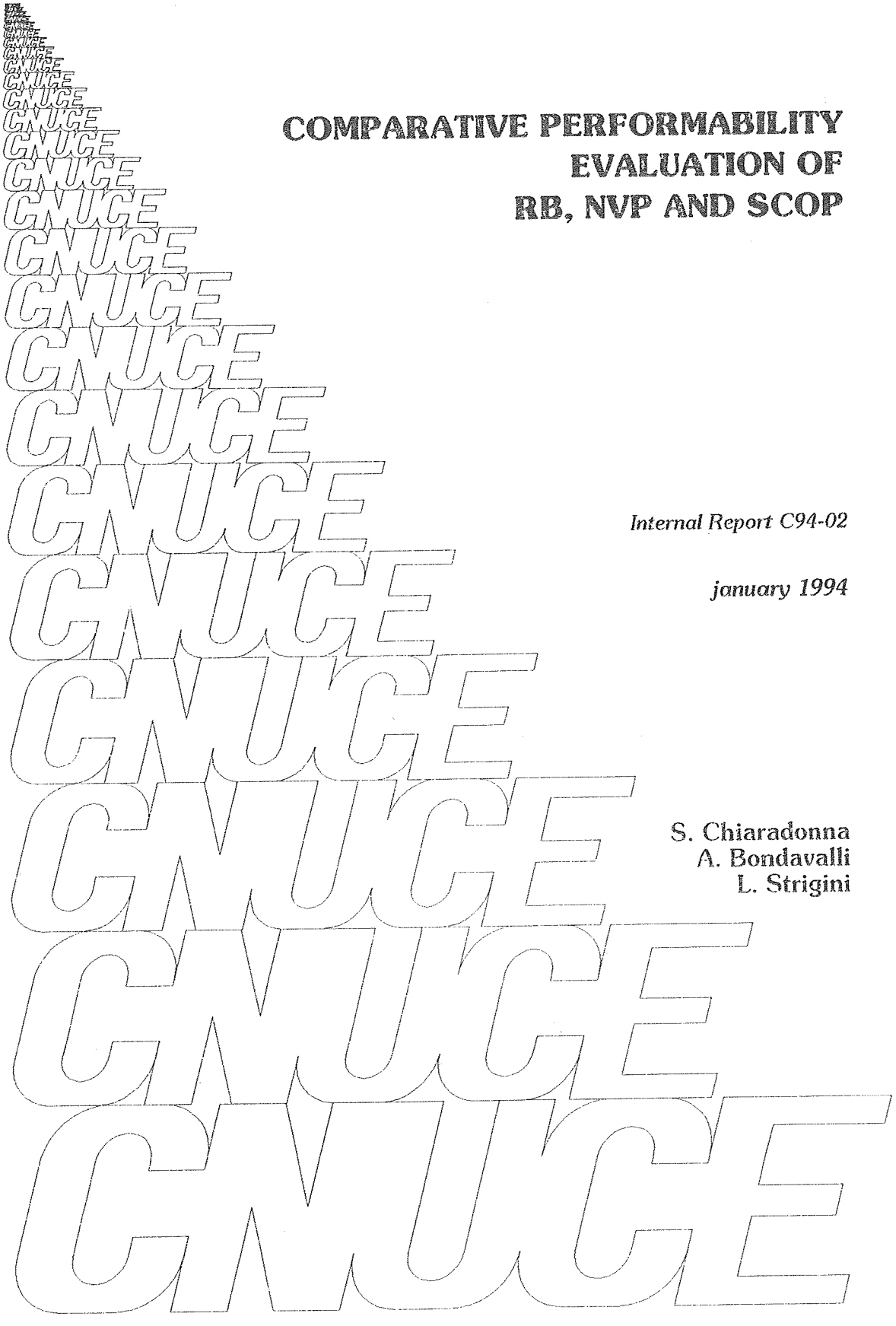



# COMPARATIVE PERFORMABILITY EVALUATION OF RB, NVP AND SCOP

*Internal Report C94-02*

*january 1994*

S. Chiaradonna
A. Bondavalli
L. Strigini

# Comparative Performability Evaluation of RB, NVP and SCOP

Sllvano Chiaradonnat, Andrea Bondavalli! and Lorenzo Strigini[2]

1 CNUCE/CNR, Via S. Maria 36, 56126 Pisa, Italy

E-mail: silvano@mammolo.cnuce.cnr.it.andrea.bondavalli@cnuce.cnr.it

2 IEI/CNR, Via S. Maria 46,56126 Pisa, Italy

E-mail: strigini@iei.pi.cnr.it

## Abstract

An adaptive scheme for software fault-tolerance is evaluated from the point of view of performability, comparing it with previously published analyses of the more popular schemes, recovery blocks and multiple version programming. In the case considered, this adaptive scheme, "Self-Configuring Optimistic Prograrnrning" (SCOP), is equivalent to N-version programming in terms of the probability of delivering correct results, but achieves better performance by delaying the execution of some of the variants until it is made necessary by an error. We discuss, by mean of an example, the application of modelling to realistic problems in fault-tolerant design.

## 1. Introduction

Software fault tolerance, that is, diverse redundancy in software design, is the only known way of tolerating residual design faults in operational software products. The evaluation of its effectiveness is the topic of numerous papers (most recently [1, 8, 9]). In this paper, we extend existing work on *performability* evaluation to cover a different, adaptive fault-tolerant scheme, and we discuss the application of modelling to realistic problems in fault-tolerant design.

In the fault-tolerant techniques we consider, a (fault-tolerant) software component consists of a set of diversely implemented, functionally equivalent *variants*, plus *adjudication* subcomponents. At each execution of the component, some subset of these subcomponents is executed, in such a way that they may check and correct each other's results. Many such execution schemes are possible. The best known are Recovery Blocks (RB) [12] and N-version programming (NVP) [2]. In the simplest form of NVP, the N variant are executed in parallel on the same input, and the adjudication consists in a more or less complex vote on their results [6]. In RB, only one variant is executed, at first, and if its result does not pass an *acceptance test*, other variants are invoked, in turn, until one passes or the available variants are used up. Clearly, these are two extremes in a range of trade-offs between consumption of "space" (level of parallelism) and "time" (elapsed time), and between the goals of low average resource consumption and low worst-case response time [3]. Many other combinations are

possible [13, 14]. We shall consider the scheme called "Self-Configuring Optimistic Programming" (SCOP) [4], which describes a useful family of such execution schemes.

In SCOP, a subset of the available variants is initially executed which would be enough to satisfy a *delivery condition* (e.g., that the result be correct given that no more than one variant fails during the whole execution of the redundant component; or that the result be correct with a minimum stated probability) if no errors occurred; if, then, errors do occur, additional variants may be executed. The adjudicator checks for the satisfaction of this delivery condition, in terms of agreement among the results produced, and then if necessary more variants are executed until either the variants are exhausted or so many errors have occurred that the delivery condition can no longer be satisfied. The scheme is thus configured by assigning the delivery condition, the number of variants available, and in addition a maximum allowable number of execution rounds (to represent real-time constraints). So, a simple example of SCOP employs three variants: if the delivery condition is that the acceptable result must have a 2-out-of-3 majority, and two rounds are allowable, then 2 variants will be executed at first, and their results accepted if in agreement, otherwise the third variant will be executed and voted with the other two. If the maximum acceptable number of rounds were 1, then the SCOP scheme would execute as parallel NVP. If the delivery condition were just that a variant produce a result that it can itself trust, and three rounds were acceptable, then SCOP would be a 3-variant recovery block.

These are only stylised descriptions of error treatment. Among the factors not described so far, we could mention that in NVP schemes error detection is likely not to depend only on voting / comparison, as most software components have some capability for self-checking (e.g., range-checking on procedure arguments, divide-by-zero checks, etc.). In most applications, some kind a watchdog timer will be employed to prevent variants from running for an inordinately long time. All these schemes describe the error treatment performed at *one* invocation of the software component, from a set of input to a set of outputs. They do not imply one choice in the organisation of longer-term execution. For instance, the variants in NVP could be organised as rather independent, long-lived processes, with adjudication only on results output to the external world, or as iterations of short execution stages, each of which produces both adjudged outputs and an adjudged initial state for the next iteration.

This paper deals with the *evaluation* of software fault tolerance schemes. The architect of a system needs to evaluate the results of employing software fault-tolerant scheme, in order both to evaluate the effects of the different available design alternatives and, once design decisions are made, to predict the overall behaviour of the system (for instance as a basis for building a case for the acceptance of the system).

Schemes for software fault tolerance can be evaluated and compared according to different figures of merit. First, the probability of delivering a correct result (or, alternatively, a safe result - including an exception signalling a detected error) at one invocation of the software, which depends on the probabilities of the different combinations of erroneous and correct execution by the various components in the redundant component. The functions of the individual application components are probably the main factor here determining, e.g., the ease of writing effective and efficient acceptance tests. With these figures one can assess components for which the main figures of merit are the probabilities of failure on demand (like some safety systems), and proceed to estimate other figures of merit. One can then inquire about the probability of a component not failing by a given time in the future (reliability predictions), or of it not producing an *undetected* failure. This category of results will typically be of interest for application such as avionics, where the main figure of merit is the survival probability over a short mission without repair. Reliability evaluations have been provided e.g. in [1, 11]. Performance is also an important deciding factor, in the form of response time and throughput [7].

In many cases, more complex probabilistic assessments of the utility (or cost) derived from operating a system are of interest (*performability* evaluation [10]). [15, 16] have proposed performability evaluations of schemes for software fault tolerance. We use and expand these results. We model and compare recovery blocks with two variants, N-version programming with three variants, and SCOP with three variants executed in 2 rounds.

It is appropriate here to define the goal of this evaluation exercise. Once a complete hardware-software system has been completely defined, a realistic performability assessment could be obtained by modelling the process of demands on the system as well as the behaviour of the software executing on the limited hardware resources available. The evaluations we give here (like those by most other authors) are not of this kind. Rather, they are derived assuming unlimited resources and infinite load. As such, they are independent of any individual application, and can be considered as limiting results useful in the early dimensioning of a design, like, for instance, figures of throughput of a communication channel with permanently full input queues.

In the next section, we describe the class of systems we plan to evaluate, with the assumptions that affect our models, and describe the modelling approach and the reward function used, which are taken from [15, 16]. In the Section 3 and 4 we describe the models for evaluating the performability of the recovery blocks and N-version programming schemes. The main contributions of this paper are: a model for evaluating the performability of the SCOP family of fault-tolerant software designs, described in Section 5, with, in Section 6, a comparison of NVP, RB and intermediate schemes such as SCOP. Throughout Sections 3, 4, 5 and 6, the assumptions are consistent with [15, 16], so as to allow a comparison of the results from SCOP with those derived there. All departures from those assumptions and their effects on the results are mentioned explicitly.

# 2. Background

## 2.1. The system

We assume here an application of an iterative nature, where a *mission* is composed of a series of iterations of the execution of the fault-tolerant software component. At each iteration, the component accepts an input and produces an output. If the execution lasts beyond a pre-set maximum duration, it is aborted by a watchdog timer. The outcomes of an individual iteration may be: i) success, i.e., the delivery of a correct result, ii) a "benign" failure of the component, i.e., a detected error (detected either by comparison of redundant results, by an acceptance test or by the watchdog timer), or iii) an undetected error ("catastrophic" failure: delivery of an erroneous result).

For this scenario, performability figures are a function of the assumed load and of the hardware resources available (processors, etc.). Instead of assuming a hypothetical load and hardware configuration, unlimited resources and an "infinite" load are assumed: the redundant component always executes with the maximum degree of parallelism allowed by its design, and as soon as an iteration is over the next iteration is started.

The reward measure used as a basis for performability evaluation is as follows: successful executions of the redundant component add one unit to the value of the mission; executions producing detected errors add zero; an undetected error reduces the value of the whole mission to zero. The accrued value over a mission is called $M_t$, and the expected value of this measure is evaluated.

Albeit unrealistic, this model can be used as a limiting case, allowing one to answer the question: if the amount of computation performed is only limited by the internal structure of the software (durations of subcomponent executions and precedence order between them), how much value can the system produce over a mission? This is a question similar to asking for the statistics of the response time for a software component, but also takes into account the different reward levels to be expected in different executions because of errors.

This model and reward function imply that each iteration needs the output of the previous one, but a detected failure of an individual iteration is assumed not to damage the mission (e.g. because the controlled system can be kept under control by supplying default safe outputs from the computer), nor to affect subsequent executions (no propagation of errors). Additional assumptions used are:

compares their results, seeking a 2-out-of-3 majority. If majority exists it accepts them (typically, it delivers one of them as the result of the redundant component). Otherwise, the result is suppressed. The paths in Figure 4 correspond to the different possible outcomes:

(1): there exists a majority representing a correct computation and the output is a correct result;

(2): an erroneous result is accepted (catastrophic failure);

(3): the result is rejected (benign failure);

(4): the duration of the redundant execution exceeds a specified limit (the real-time constraint) and the execution is aborted (benign failure).

### 4.2. The Dependability Submodel for NVP

The relevant events defined on the outcomes of one execution of the NVP component and the notation for their probabilities are as illustrated in Table 3. As for RB, the assumption of no compensation between errors has allowed us to reduce the event space to be considered.

| Error Types | Probabilities |
|---|---|
| 3 variants err with consistent results | $q_{3v}$ |
| 2 variants err with consistent results (the 3rd result is inconsistent with them, and may be correct or erroneous) | $q_{2v}$ |
| The adjudicator errs by selecting an erroneous, non-majority result | $q_{vd}$ |
| A variant errs, conditioned on none of the above events happening (i.e., there are one or more *detected* errors; their statistical independence is assumed) | $q_{iv}$ |
| The adjudicator errs by not recognising a majority (hence causing a benign failure), conditioned on the existence of a majority | $q_d$ |

**Table 3: Error Types and Notation for NVP**

The detailed model of one execution of the redundant component, without considering the operation of the watchdog timer, is shown in Figure 5. Table 4 shows the definitions of the states. The graph shows how certain executions terminate. In practice, it will later be apparent that some of the parameters describing the model have little influence on the solution.

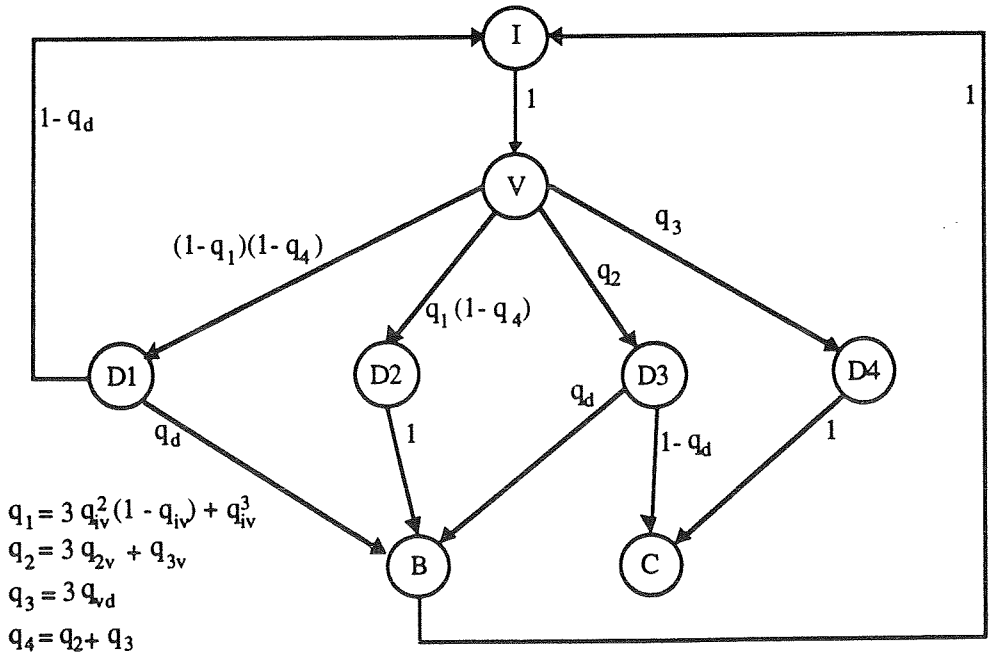



**Figure 5: The Dependability Submodel for NVP**

| States | Definition |
|---|---|
| $I$ | initial state of an iteration |
| $V$ | execution of versions |
| $\{D_i \mid i \in \{1,2,3,4\}\}$ | execution of decision function |
| $B$ | benign error (caused by a detected value error) |
| $C$ | catastrophic error (caused by an undetected value error) |

**Table 4: State Definitions for NVP Dependability Model**

Now we briefly describe the meanings of the arcs from V. The descriptions are out of numerical sequence to simplify the explanation.

$D_3$: there exists a majority representing an erroneous result (2 or 3 variants are erroneous and in agreement); this leads to either a catastrophic or a benign failure, depending on whether the adjudicator recognises this majority or fails to recognise it;

$D_4$: one or more variants err with inconsistent results and the decision function accepts an inconsistent erroneous result (this leads to a catastrophic failure);

$D_2$: none of the above events occurs, and two or three variants err with inconsistent results (this leads to a benign failure);

$D_1$: none of the above events occurs, and there exists a majority representing a correct result; this leads to either a success or a benign failure, depending on whether the adjudicator recognises this majority or fails to recognise it.

To simplify the expression of the solution, we define a set of intermediate parameters as shown in the bottom left corner of Figure 5. We call the probabilities of a catastrophic and of a benign failure, without the watchdog timer (that is, due solely to the *values* of the results of the subcomponents), $p_{cv}$ and $p_{bv}$, respectively. From the state transition diagram, it follows that:

$$p_{cv} = 3q_{vd} + q_2(1-q_d)$$

$$p_{bv} = (1-q_1)(1-q_4)q_d + q_1(1-q_4) + q_2q_d =$$
$$= (1-q_2)q_d + q_1(1-q_2)(1-q_d) + q_2q_d - 3q_{vd}[q_d + (1-q_d)q_1].$$

## 4.3. The Performance Submodel for NVP

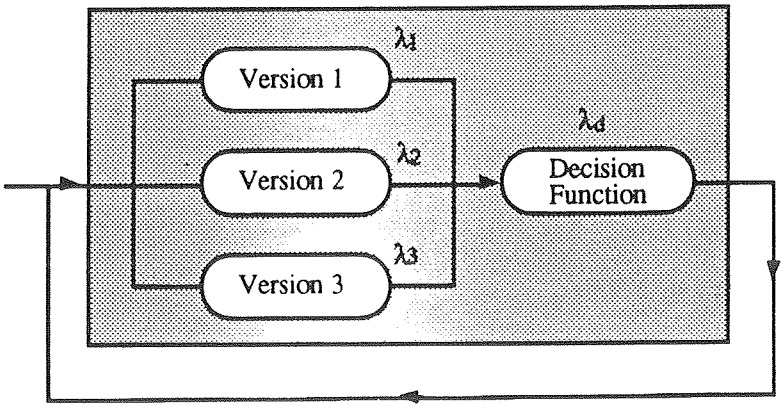



**Figure 6: The Performance Submodel for NVP**

To model the performance of NVP we adopt the same construction of [15]. The assumptions here are that the execution times for the three variants and the adjudicator, called $Y_1$, $Y_2$, $Y_3$ and $Y_d$, are independently and exponentially distributed, with parameters $\lambda_1$, $\lambda_2$ , $\lambda_3$ and $\lambda_d$. They are also assumed to be independent of the events considered in the dependability submodel. The maximum execution time allowed by the watchdog timer is called $\tau$. We designate $Y_c$ and Y, respectively, the duration of an execution of the redundant component if

the watchdog timer is absent and if it is present. For our purposes, it is sufficient to compute the mean $\mu$ and variance $\sigma^2$ of the distribution of Y and the probability $p_{bt}$ that an execution violates the timing constraint (that is, $Y_c$ exceeds $\tau$). Figure 6, derived from Figure 4 (NVP operation), depicts this performance submodel.

If we designate $Y_v$ the parallel execution time of the three variants, $Y_v = \max \{Y_1, Y_2, Y_3\}$, its cumulative distribution function is easily obtained as:

$$G_{Y_v}(y) = \begin{cases} (1 - e^{-y\lambda_1})(1 - e^{-y\lambda_2})(1 - e^{-y\lambda_3}), & \text{if } y \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

Using the Laplace transforms of the density functions, to avoid convolutions in our expressions, and denoting the Laplace transforms of the probability density functions of $Y_v$ and $Y_d$ as $L_{Yv}$ and $L_{Yd}$ respectively, the transform of the probability density functions of $Y_c$ is:

$L_{Yc}(s) = L_{Yv}(s)\, L_{Yd}(s)$.

Through the inverse Laplace transform we then obtain the probability density function of $Y_c$, and then, considering that all the executions that would last more than $\tau$ without the watchdog last exactly $\tau$ with the watchdog, that of Y in a manner analogous to that of the RB solution (Section 3.3).

We now compute the probabilities that the execution completes with catastrophic failure, benign failure or success, denoted as $p_c$, $p_b$ and $p_{succ}$ respectively, considering that the intervention of the watchdog timer turns into benign failures some executions which would otherwise produce success or catastrophic failure (here we depart from the procedure of [15, 16]). In the Section 4.2, from Figure 5 we have derived the probabilities of benign value failures ($p_{bv}$), catastrophic value failures ($p_{cv}$) and success ($1 - p_{bv} - p_{cv}$) representing the possible outcomes of the scheme whose executions are stopped by the watchdog timer. According to the assumption of independence between the execution times of the subcomponents and their error behaviours we derive the following probabilities (which are not fully developed here for sake of brevity) for executions of the scheme with the watchdog timer:

$p_b = p_{bt} + p_{bv} - p_{bt}\, p_{bv}$;

$p_c = p_{cv} - p_{bt}\, p_{cv}$;

$p_{succ} = (1 - p_b - p_c)$.

### 4.4. Performability

As shown in [15] the performability model for NVP is similar to the corresponding RB model. Thus, the performability measure $E[M_t]$ is obtained via the same general equation of the RB model (Section 3.4) after substituting the information supplied by the dependability and performance submodels of NVP.

## 5. The SCOP model

### 5.1. Operation of SCOP

A redundant component based on the SCOP scheme with 3 variants includes:

- three functionally equivalent but independently developed programs (variants);
- an adjudicator which determines a consensus result from the results delivered by the variants. We assume as a delivery condition a 2-out-of-3 majority (equivalent to correctness in the presence of at most one error by a variant);

- a watchdog timer which detects violations of the timing constraint (executions exceeding the maximum allowed duration).

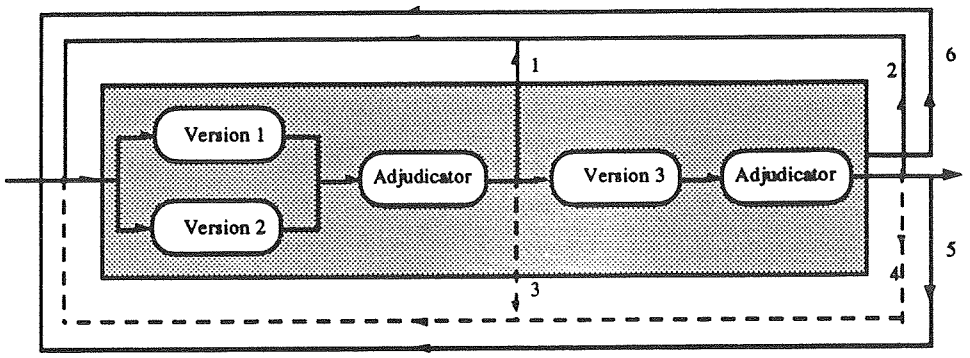



**Figure 7: SCOP Operation**

Figure 7 shows the operation of the SCOP scheme. Each iteration is divided in two phases. In the first phase, variant 1 and variant 2 begin to execute at the same time. After both have completed their executions, the adjudicator compares their results. If they are consistent, it accepts them (typically, it delivers one of them as the result of the redundant component). Otherwise, the second phase begins, variant 3 executes, and then the adjudicator decides on the basis of all three results, seeking a 2-out-of-3 majority. The paths in Figure 7 correspond to the different possible outcomes:

(1): at the end of the first phase there exists a majority representing a correct computation and the output is a correct result;

(2): at the end of the first phase the result is rejected, at the end of the second phase there exists a majority representing a correct computation and the output is a correct result;

(3): at the end of the first phase an erroneous result is accepted (catastrophic failure);

(4): at the end of the first phase the result is rejected, at the end of the second phase an erroneous result is accepted (catastrophic failure);

(5): at the end of the second phase the result is rejected (benign failure);

(6): the duration of the redundant execution exceeds a specified limit (the real-time constraint) and the execution is aborted (benign failure).

## 5.2. The Dependability Submodel for SCOP

The relevant events defined on the outcomes of one execution of the SCOP component and the notation for their probabilities are as illustrated in Table 5. The assumption of no compensation between errors has allowed us to reduce the event space to be considered.

| Error Types (Events) | Probabilities |
|---|---|
| 3 variants err with consistent results | $q_{3v}$ |
| 2 variants err with consistent results (the 3rd result is inconsistent with them, and may be correct or erroneous) | $q_{2v}$ |
| The adjudicator errs and terminates the execution with phase 1, selecting an erroneous, non-majority result | $q_{vd1}$ |
| The adjudicator errs and terminates the execution with phase 2, selecting an erroneous, non-majority result | $q_{vd2}$ |
| A variant errs, conditioned on none of the above events happening (i.e., there are one or more detected errors; their statistical independence is assumed) | $q_{iv}$ |
| The adjudicator errs, at the end of either phase 1 or phase 2, by not recognising a majority (hence causing a benign failure), conditioned on the existence of a majority | $q_{id}$ |

**Table 5: Error Types and Notation for SCOP**

The detailed model of one execution of the redundant component, without considering the operation of the watchdog timer, is shown in Figure 8. Table 6 shows the definitions of the states. The graph is somewhat complex, in order to represent clearly all the possible paths of execution, showing how certain executions terminate with the first phase, while others go on with the execution of the third variant and a new adjudication. In practice, it will later be apparent that some of the parameters describing the model have little influence on the solution.

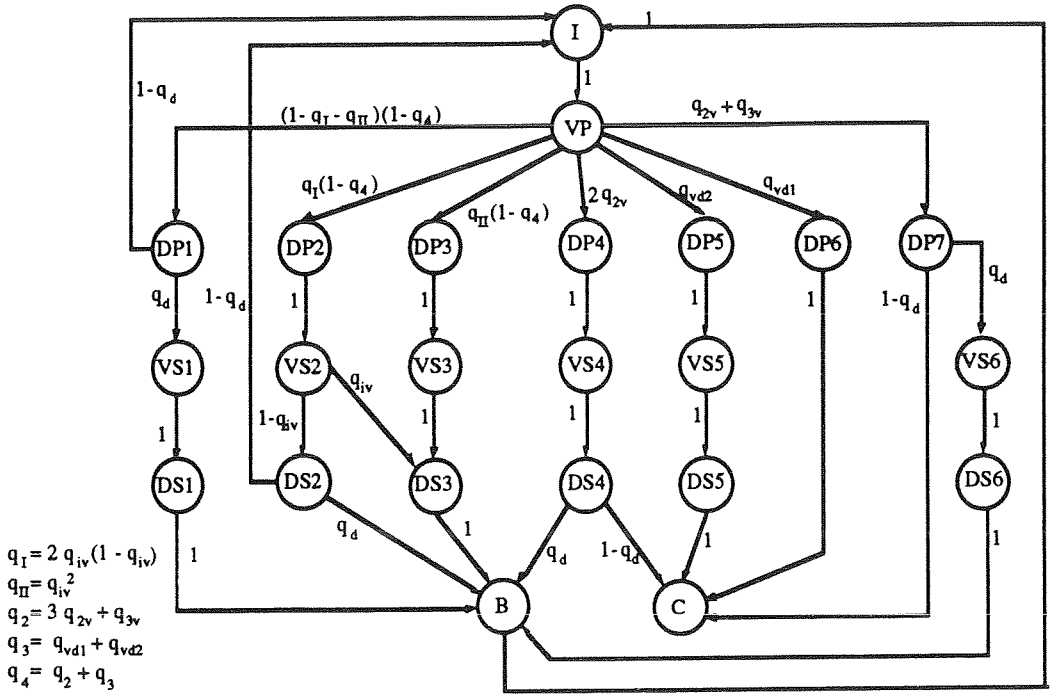



**Figure 8: The Dependability Submodel for SCOP**

| States | Definition |
|---|---|
| *I* | initial state of an iteration |
| *VP* | execution of two variants in the first phase |
| $\{DP_i \mid i \in \{1,2,3,4,5,6,7\}\}$ | execution of adjudicator after *VP* |
| $\{VS_i \mid i \in \{1,2,3,4,5,6\}\}$ | execution of one variant in the second phase |
| $\{DS_i \mid i \in \{1,2,3,4,5,6\}\}$ | execution of adjudicator after $VS_i$ |
| *B* | benign failure (caused by a detected value error) |
| *C* | catastrophic failure (caused by an undetected value error) |

**Table 6: State Definitions for SCOP Dependability Model**

Most of our parameters are the inconditional probabilities of sets of outcomes of the whole redundant execution (including the executions of both the variants and the adjudicator): hence, some of the arcs exiting VP are labelled with these probabilities, and are followed by arcs, as e.g. from $DP_5$ to $VS_5$, labelled with a probability equal to 1.

We briefly describe the meanings of the arcs from VP. The descriptions are out of numerical sequence to simplify the explanation.

$DP_7$: at the end of phase 1, variants 1 and 2 are both erroneous and in agreement (this includes the case of a consistent error among all 3 variants, an event which has a clear physical meaning - presentation of an input on which all 3 variants would fail with consistent results -, though it can only be observed if the adjudicator fails to recognise the agreement in phase 1);

$DP_5$: variants 1 and 2 are correct and thus in agreement, variant 3 fails, and the adjudicator fails in such a way as not to recognise the agreement in phase 1, and to choose the result of variant 3 as a majority;

$DP_6$: one among variants 1 and 2 fails, and there is not majority, but the adjudicator fails to notice the disagreement and chooses the wrong result as correct;

$DP_4$: at the end of phase 1 there is no majority (either one variant is in error, or both are, but with inconsistent results), and then variant 3 also errs, forming an erroneous majority with either variant 1 or variant 2. Neither $DP_5$ nor $DP_6$ occurs. This leads to either a catastrophic or a benign failure, depending on whether the adjudicator recognises this majority or fails to recognise it;

$DP_3$: none of the above events occurs, and variants 1 and 2 produce inconsistent, erroneous results: no majority exists; the adjudicator recognises the lack of a majority;

$DP_2$ : none of the above events occurs; one among variants 1 and 2 produces an erroneous result; depending on whether variant 3 produces a correct result, phase 2 terminates with a correct majority ($DS_2$) or not ($DS_3$);

$DP_1$: none of the above events occurs; variants 1 and 2 are correct.

In states $DP_1$, $DS_2$, $DS_4$, $DP_7$ a majority exists. The adjudicator may fail to recognise it, with probability $q_d$, and produce a benign failure. It has been plausibly assumed that if the adjudicator fails in this fashion at the end of phase 1, it will consistently fail at the end of phase 2: hence the probabilities equal to 1 on the arcs downstream of $DS_1$ and $DS_6$.

To simplify the expression of the solution, we define a set of intermediate parameters as shown in the bottom left corner of Figure 8. We call the probabilities of a catastrophic and of a benign failure, without the watchdog timer (that is, due solely to the *values* of the results of the subcomponents), $p_{cv}$ and $p_{bv}$, respectively. From the state transition diagram, it follows that:

$$p_{cv} = q_3 + q_2(1-q_d)$$

$$p_{bv} = q_2 q_d + q_{II}(1-q_4) + q_I(1-q_4)(q_{iv} + (1-q_{iv})q_d) + (1-q_I-q_{II})(1-q_4)q_d =$$

$$= (1-q_4)q_d + q_I(1-q_4)(1-q_d) + q_2 q_d$$

These expressions are quite similar to those that we obtained for N-version programming, as will be discussed later. The NVP and SCOP schemes behave instead quite differently from the point of view of performance.

The expressions for $p_{bv}$ and $p_{cv}$ for SCOP only differ from those obtained for NVP in having $q_{vd1}+q_{vd2}$ instead of 3 $q_{vd}$. When evaluating these expressions in the next section, we have rather arbitrarily considered both $q_{vd2}$ and $q_{vd1}$ as "common-mode failures among the adjudicator and one variant", and accordingly assigned them the probabilities $q_{vd}$ and 2 $q_{vd}$, respectively: with the values we have later assigned to these parameters, this arbitrary assignment has a negligible effect on the results. The NVP and SCOP schemes behave in exactly the same manner with regard to failures of the variants: a SCOP execution scheme using 2-out-of-3 majority guarantees that, with a correct adjudicator, exactly the same outcome will be produced as that produced by the same variants organised in an NVP scheme. The differences may lay in the error behaviour of the adjudicator, and the different probabilities of the outcomes involving such errors. These probabilities are exceedingly difficult to estimate, but in the next section we plausibly assume them to be low compared to those of one or two variants failing. If, however, this assumption were not verified, deriving such probabilities would be quite difficult.

### 5.3. The Performance Submodel for SCOP

The assumptions here are that the execution times for the three variants and the adjudicator, called $Y_1$, $Y_2$, $Y_3$ and $Y_d$, are independently and exponentially distributed, with parameters $\lambda_1$, $\lambda_2$ , $\lambda_3$ and $\lambda_d$ (the execution durations of the adjudicator at the first and second phase are drawn from the same distribution). They are also assumed to be independent of the events considered in the dependability submodel. The maximum execution time allowed by the watchdog timer is called $\tau$. We designate $Y_c$ and Y, respectively, the duration of an execution

of the redundant component if the watchdog timer is absent and if it is present. For our purposes, it is sufficient to compute the mean $\mu$ and variance $\sigma^2$ of the distribution of Y and the probability $p_{bt}$ that an execution violates the timing constraint (that is, $Y_c$ exceeds $\tau$).

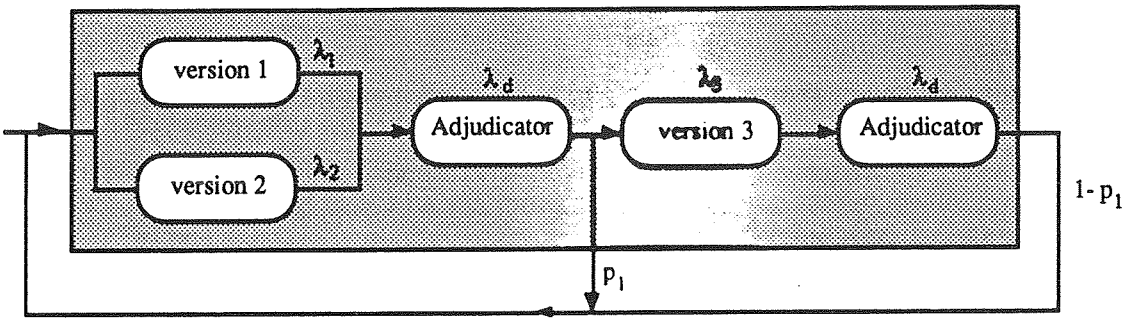



**Figure 9: The Performance Submodel for SCOP**

Figure 9, derived from Figure 7 (SCOP operation), depicts this performance submodel. The path labelled "$p_1$" corresponds to paths 1 and 3 in Figure 7; $p_1$ is the probability that an execution completes at the end of the first phase. The path labelled "$1-p_1$" corresponds to paths 2, 4 and 5, and $1-p_1$ is the probability that the execution includes phase 2. From Figure 8 (dependability model) we derive the probability $p_1$:

$$p_1 = (1-q_I-q_{II})(1-q_4)(1-q_d)+(q_{2v}+q_{3v})(1-q_d)+2q_{vd}.$$

If we designate $Y_v$ the parallel execution time of the first two variants, $Y_v = \max\{Y_1, Y_2\}$, the execution time $Y_c$ without the watchdog timer is:

$$Y_c = \begin{cases} Y_{c1} = Y_v + Y_d = \max\{Y_1, Y_2\} + Y_d, & \text{with probability } p_1 \\ Y_{c2} = Y_v + Y_d + Y_3 + Y_d = \max\{Y_1, Y_2\} + Y_3 + 2Y_d, & \text{with probability } (1-p_1) \end{cases}$$

The probability density function of $Y_c$ denoted as $f_{Yc}$ is a weighted sum of the probability density functions for the two expressions above. The only random variable in these expressions that is not exponentially distributed is $Y_v$, whose cumulative distribution function is easily obtained as:

$$G_{Y_v}(y) = \begin{cases} (1 - e^{-y\lambda_1})(1 - e^{-y\lambda_2}), & \text{if } y \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

Using the Laplace transforms of the density functions, to avoid convolutions in our expressions, and denoting the Laplace transforms of the probability density functions of $Y_v$, $Y_d$ and $Y_3$ as $L_{Yv}$, $L_{Yd}$ and $L_{Y3}$ respectively, the transform of the probability density functions of $Y_c$ is:

$$L_{Yc}(s) = p_1 L_{Yv}(s) L_{Yd}(s) + (1-p_1) L_{Yv}(s) L_{Yd}(s) L_{Y3}(s) L_{Yd}(s)$$

From this we can compute the probability density function of $Y_c$, and then, considering that all the executions that would last more than $\tau$ without the watchdog last exactly $\tau$ with the watchdog, that of Y:

$$f_Y(y) = \begin{cases} f_{Yc}(y), & \text{if } y < \tau \\ p_{bt}\,\delta(y-\tau), & \text{if } y \geq \tau \end{cases}$$

where $\delta(y-\tau)$ is the unit impulse function, and $p_{bt} = 1 - \int_0^\tau f_{Yc}(y)dy$. The mean and variance of Y will be used directly in the performability calculations.

We now compute the probabilities that the execution completes with catastrophic failure, benign failure or success, denoted as $p_c$, $p_b$ and $p_{succ}$ respectively, considering that the intervention of the watchdog timer turns into benign failures some executions which would otherwise produce success or catastrophic failure (here we depart from the procedure of [15, 16]).

As for RB scheme, we divide the probabilities of exceeding the maximum allowed duration among the different outcomes possible at each phase. We can write $p_{bt}$ as $p_{bt}=p((v1 \wedge Yc1>\tau) \vee (v2 \wedge Yc2>\tau))$, where v1 is the event 'only one phase needed' and has probability $p_1$ while v2 represent the event 'two phases necessary' with probability $(1 - p_1)$. According to the assumption of independence between the execution times of the subcomponents and their error behaviours:

$$p_{bt} = p(v1)\, p(Y_{c1} > \tau) + p(v2)\, p(Y_{c2} > \tau).$$

Both v1 and v2 can be split again in disjoint sub-events representing the possible outcomes of the scheme when execution ends in phase one or continues to phase 2. Using the same notation of RB (Section 3.3) we obtain:

$$p_{bt} = (p_{s1} + p_{bv1} + p_{cv1})p(Y_{c1} > \tau) + (p_{s2} + p_{bv2} + p_{cv2})p(Y_{c2} > \tau).$$

From Figure 8 (dependability submodel) we derive all these probabilities:

$$p_{bv1} = 0,$$

$$p_{bv2} = p_{bv},$$

$$p_{cv1} = (q_{2v} + q_{3v})(1 - q_d) + q_{vd1},$$

$$p_{cv2} = 2q_{2v}(1 - q_d) + q_{vd2},$$

$$p_{s1} = (1 - q_I - q_{II})(1 - q_4)(1 - q_d),$$

$$p_{s2} = q_I(1 - q_4)(1 - q_{iv})(1 - q_d).$$

Now we can regroup the previous expression of $p_{bt}$ to make explicit which of the executions stopped by the watchdog timer would have ended with success, catastrophic value failures or benign value failures. Finally, applying the same procedures of RB scheme (Section 3.3) we obtain the following probabilities (which are not fully developed here for sake of brevity) for executions of the scheme with the watchdog timer:

$$p_b = p_{bt} + p_{bv} - p_{bv1}\, p(Y_{c1} > t) - p_{bv2}\, p(Y_{c2} > t);$$

$$p_c = p_{cv} - p_{cv1}\, p(Y_{c1} > \tau) - p_{cv2}\, p(Y_{c2} > \tau);$$

$$p_{succ} = (1 - p_b - p_c).$$

### 5.4. Performability

As shown in [15] the performability model for SCOP is similar to the corresponding RB model. Thus, the performability measure $E[M_t]$ is obtained via the same general equation of the RB model (Section 3.4) after substituting the information supplied by the dependability and performance models of SCOP.

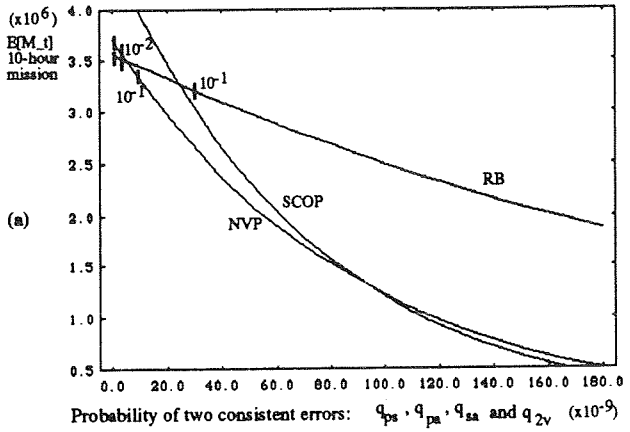
## 6. Evaluation results

We now show the results obtained from the models described above. Initially, we plot (Figure 10) the solutions of the models for exactly the same parameter values used in [15] and reported in Table 7, so as to allow a direct comparison.

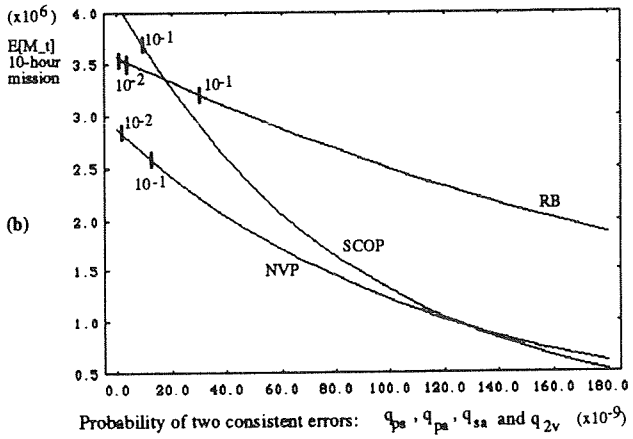
The issue arises here of the values and ranges chosen for these parameters. Consistent errors of more than one variant are plausibly the main factor in determining the outcome of the individual executions: it is therefore interesting to plot the variation in performability obtained by varying

this probability, from being negligible to being much higher than the other error probabilities, while keeping all others constant. In all the figures, the mission duration is 10 hours, a reasonable order of magnitude for e.g. a workday in non-continuous process factory operation, or in office work, and flight duration for civil avionics.

| Timing parameters ($msec^{-1}$) |
|---|
| Recovery Blocks |
| $\lambda_p=\lambda_s=\lambda_a=$ $=1/5$ $\tau=30$ msecs |
| NVP and SCOP |
| $\lambda_1=\lambda_2=\lambda_3=$ $=1/5$ $\lambda_d=2$ $\tau=30$ msecs |




| Timing parameters (msec -1) |
|---|
| Recovery Blocks |
| $\lambda_p = 1/5$ $\lambda_s = 1/8$ $\lambda_a = 1/5$ $\tau = 30$ msecs |
| N-version programming and SCOP |
| $\lambda_1 = 1/5$ $\lambda_2 = 1/6$ $\lambda_3 = 1/8$ $\lambda_d = 2$ $\tau = 30$ msecs |




| Timing parameters ($msec^{-1}$) |
|---|
| Recovery Blocks |
| $\lambda_p = 1/5$ $\lambda_s = 1/18$ $\lambda_a = 1/5$ $\tau=30$ msecs |
| NVP and SCOP |
| $\lambda_1 = 1/5$ $\lambda_2 = 1/6$ $\lambda_3 = 1/18$ $\lambda_d = 2$ $\tau=30$ msecs |

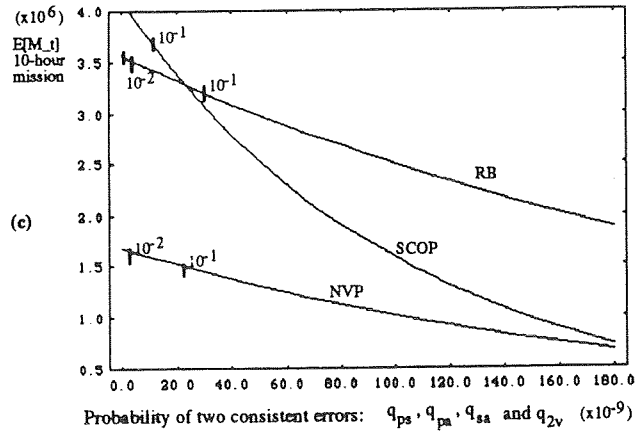



**Figure 10: Comparisons of the RB, NVP and SCOP schemes. As shown in the three tables, the execution rates of the variants are equal in (a), and become progressively more different in the other two figures.**

| Recovery Blocks | N-Version Programming | SCOP |
|---|---|---|
| $q_{ps} = q_{pa} = q_{sa}$: variable from 0 to $1.8\ 10^{-7}$<br>$q_{psa} = 10^{-10}$<br>$q_p = 10^{-4}$<br>$q_s = 10^{-4}$<br>$q_a = 10^{-4}$ | $q_{2v}$ :variable from 0 to $1.8\ 10^{-7}$<br>$q_{3v} = 10^{-10}$<br>$q_{iv} = 10^{-4}$<br>$q_{vd} = 10^{-10}$<br>$q_d = 10^{-9}$ | $q_{2v}$ :variable from 0 to $1.8\ 10^{-7}$<br>$q_{3v} = 10^{-10}$<br>$q_{iv} = 10^{-4}$<br>$q_{vd1} = 2\ 10^{-10}$<br>$q_{vd2} = 10^{-10}$<br>$q_d = 10^{-9}$ |

**Table 7: Values of the "dependability" parameters used for Figure 10**

The values assigned to the other parameters reflect some plausible assumptions: the adjudicator (acceptance test for recovery blocks) has a much lower error probability than the variants in an NVP or SCOP system, and a comparable probability for recovery blocks; the probabilities of coincident errors of three subcomponents are significantly lower than those of two independent (and detectable) errors, but higher than those of three independent errors. The limits of using "plausible" values are again discussed in [5]. For the execution times, three situations are chosen: similar distributions for the three variants, slightly dissimilar distributions and strongly different distributions. $\lambda_1$ ($\lambda_p$ for the recovery block scheme) is kept constant throughout, and the other parameters ($\lambda_2$ and $\lambda_3$, or $\lambda_s$) are made smaller (longer execution times). For recovery blocks and SCOP, it is assumed that the slower variants are designated for conditional execution when errors are detected.

The marks on the curves in Figures 10.a, 10.b and 10.c indicate, for each scheme, the values of the abscissa where the probability of having at least one undetected error in a mission, which increases towards the right in the figures, exceeds certain indicative values. Their function is to mark ranges of realistic parameter values. As shown, this choice of parameters implies an exceedingly low probability of completing a mission without catastrophic failures. It seems unlikely that a developer would go to the expense of implementing diverse software for such a poor return. Mean times to failure better than hundreds of hours are well within the reach of good software development practice for non-redundant software in many applications. For the typical, critical applications of software fault tolerance, the interesting range for the abscissa would be very close to the zero.

To clarify this issue, we recall that the probability of surviving a mission with n iterations is $(1-p_c)^n$, so that it decreases dramatically with increasing numbers of executions per mission, even for comparatively low values of $p_c$, as plotted in Figure 11. Our 10-hour mission includes a few millions of executions.[1]

However, the curves do show the important factors in the behaviour of the models. The performability measure $E[M_t]$ is approximately equal to the product of the following factors:

1. the expected number of executions in a mission. From this point of view, having to wait for the slower among two variants (for SCOP) or the slowest among three (for NVP) is a disadvantage. So, RB performs better than SCOP, and SCOP performs better than NVP. However, the adjudication has been assumed to be faster for SCOP and NVP than for RB, and this explains the high values shown for SCOP for abscissa close to zero. The number of executions is also affected by the fact that an execution may, in RB and SCOP, last for two phases instead of one; but this may only affect the small fraction of executions where at least one error takes place, so that the number of executions per mission can be considered practically constant, for a given scheme, once the distributions of execution times for the subcomponents are assigned;
2. the probability of completing a mission without a catastrophic failure, determined by the probability of catastrophic failure per execution, $p_c$, which is, in most of the plot, practically equal to the probability of two variants delivering consistent erroneous

[1] A further warning applies here: these survival probabilities are probably highly pessimistic, as we have assumed independence between the outcomes of successive executions. This assumption is inherent in the model, so we cannot relax it here. Its consequences are discussed in [5].

results (for NVP and SCOP), or, in the recovery block scheme, of the primary variant producing an erroneous result accepted by the acceptance test. This determines how many of the missions yield a utility greater than 0;

3. the probability of benign failures, which in these plots is practically constant (for each scheme), and determines the fractions of executions in a mission which contribute to the utility of the mission.

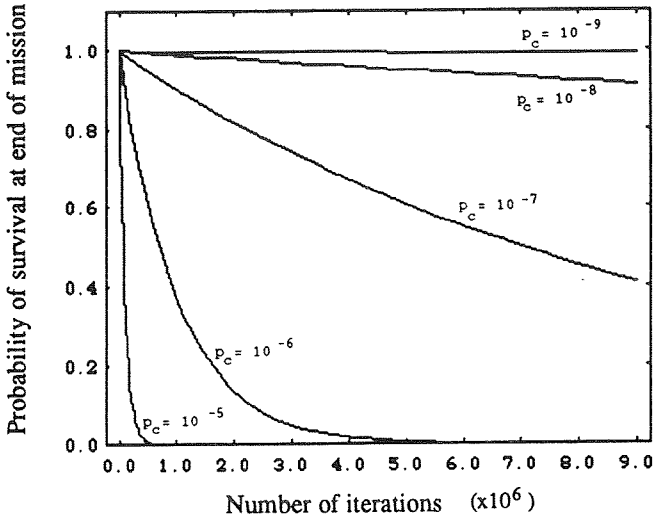



**Figure 11: Probability of survival at end of mission**

These considerations explain the shape of the plots shown. Towards the left in these figures, as the probability of catastrophic failure approaches zero, the utility of a mission tends to the mean number of executions in a mission, decreased by the (constant) fraction of benign failures. The advantages of SCOP and RB described in point 1 above predominate. As one moves to the right, the probability of catastrophic failures, and hence missions with zero utility, increases. SCOP and RB, being able to pack more execution in the same mission time, suffer more than NVP. The differences among the three figures are explained by considering that differences in the mean execution times of the variants increase the performance disadvantage of NVP with respect to SCOP, and of SCOP with respect to recovery blocks. With our parameters, while the number of executions per mission is maximum in SCOP, which explains SCOP having the highest $E[M_t]$ for the lower values of the abscissa, the slope of the curves is lowest for RB, as its probability of catastrophic failure per execution is roughly one third of that of the others.

An interesting consideration is that in the left-hand part of these plots, SCOP yields the best performability values, while its probability of surviving a mission is the worst. The importance of surviving a mission can determine a separate minimum requirement, in which case an evaluation based on only one of the two figures could be misleading, or be represented only by the cost assigned to failed missions. Increasing this cost would make all the curves steeper.

Since most of the range of the abscissa in these figures corresponds to high probabilities of missions with catastrophic failures, let us make some considerations about more realistic scenarios. So long as this model applies, requiring a probability of undetected errors (over a mission) low enough for critical applications implies requiring minuscule probabilities of error per execution. The effect of errors on performability would be minimal. A designer would be interested first in obtaining the required low probability of catastrophic failure, and could then predict $E[M_t]$ simply using a performance submodel.

An alternative scenario is a comparatively non-critical application. Let us assume for instance that a somewhat complex transaction-processing or scientific application is built with software fault tolerance, and with a requirement of one undetected error every 100 work days or so (requiring a costly roll-back and rerun of the transactions at the end of the day, after some inconsistency has been detected by external means). If we assume execution times in the order

of 100 times those in the previous scenario, but keep the same values for the parameters representing error probabilities, requirements of this order of magnitude are satisfied. The performability figures are then dominated by the performance factor, as shown in Figure 12.

Instead of considering the probability of "mission survival" separately, one can include it in the reward model. If one assigns a value of -200,000 (a loss exceeding the value of a typical successful mission) to a failed mission, the results vary as indicated by the lines whose label is prefixed with the '*' in Figure 12. The different slopes in these curves again show the effect of the different numbers of executions per mission and probabilities of catastrophic failure per execution.

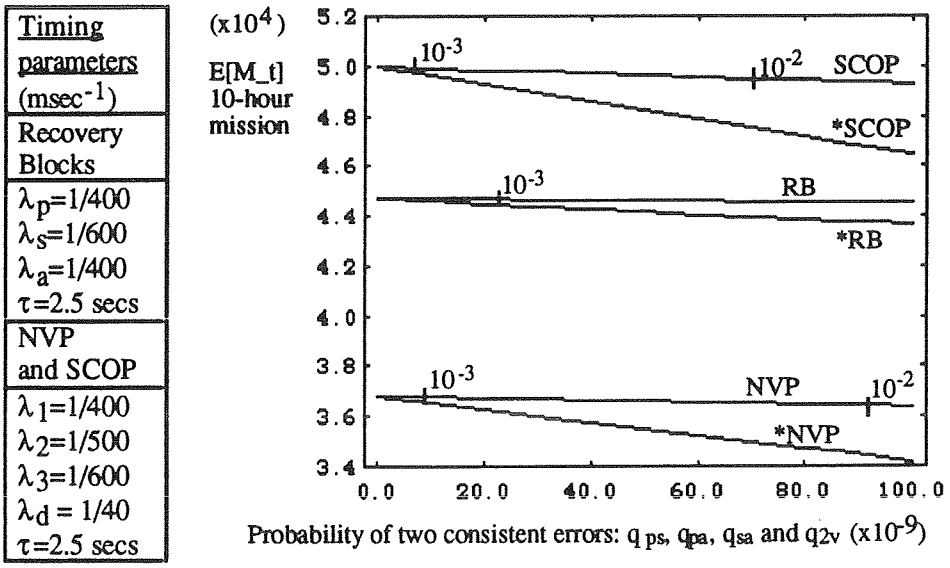



**Figure 12: Performability comparison of RB, NVP and SCOP with longer execution times**

## 7. Conclusions

We have applied an evaluation method, previously proposed by other authors, to the adaptive software fault tolerance scheme SCOP. SCOP, by delaying the execution of some variants until it is made necessary by errors, has not only a lower run-time cost than N-version programming with the same number of variants, but also a shorter response time due to a lower synchronisation overhead. The probabilities of failure per execution are the same as in NVP. The same short response time (and a better one in case of error) would be obtained by using threshold voting, as in the scheme called "NVP with tie-breaker" in [16], but without the low run-time cost. With respect to RBs, SCOP allows good error detection in applications where satisfactory acceptance tests are not available, while keeping the advantage of delaying redundant executions whenever possible, which gives it a performance and performability edge over NVP.

This kind of modelling can indicate bounds on the usability of the different schemes for software fault tolerance, subject to assumptions about the parameters. Rough estimates of the probability of failure per execution, performability measures, response time, run-time cost can together help a designer in a first choice of schemes for a design. The reward model for performability is easily changed as appropriate for different applications.

## References


[1] J. Arlat, K. Kanoun and J.C. Laprie, "Dependability Modelling and Evaluation of Sofware Fault-Tolerant Systems," IEEE TC, Vol. C-39, pp. 504-512, 1990.

[2] A. Avizienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault Tolerance During Program Execution," in Proc. COMPSAC 77, 1977, pp. 149-155.

[3] A. Avizienis and J.P.J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," IEEE Computer, Vol. 17, pp. 67-80, 1984.

[4] A. Bondavalli, F. Di Giandomenico and J. Xu, "A Cost-Effective and Flexible Scheme for Software Fault Tolerance," Journal of Computer Systems Science and Engineering, Vol. 8, pp. 234-244, 1993.

[5] S. Chiaradonna, A. Bondavalli and L. Strigini, "On Performability Modelling and Evaluation of Software Fault Tolerance Structures," 1994 (in preparation).

[6] F. Di Giandomenico and L. Strigini, "Adjudicators for Diverse Redundant Components," in Proc. SRDS-9, Huntsville, Alabama, 1990, pp. 114-123.

[7] A. Grnarov, J. Arlat and A. Avizienis, "On the Performance of Software Fault-Tolerance Strategies," in Proc. FTCS-10, Kyoto, Japan, 1980, pp. 251-253.

[8] IEEE-TR, "Special Issue on Fault-Tolerant Sofware," Vol. R-42, July, 1993.

[9] J.C. Laprie, J. Arlat, C. Beounes and K. Kanoun, "Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures," IEEE Computer, Vol. 23, pp. 39-51, 1990.

[10] J. F. Meyer, "On evaluating the performability of degradable computing systems," IEEE TC, Vol. C-29, pp. 720-731, 1980.

[11] G. Pucci, "On the Modelling and Testing of Recovery Block Structures," in Proc. FTCS-20, Newcastle upon Tyne, U.K., 1990, pp. 353-363.

[12] B. Randell, "System Structure for Software Fault Tolerance," IEEE TSE, Vol. SE-1, pp. 220-232, 1975.

[13] L. Strigini, "Software Fault Tolerance," PDCS ESPRIT Basic Research Action Technical Report No. 23, July 1990.

[14] G. F. Sullivan and G. M. Masson, "Using Certification Trails to Achieve Software Fault Tolerance," in Proc. FTCS-20, Newcastle-upon-Tyne, U.K., 1990, pp. 423-431.

[15] A. T. Tai, A. Avizienis and J. F. Meyer, "Evaluation of fault tolerant software: a performability modeling approach," in "Dependable Computing for Critical Applications 3", C. E. Landwher, B. Randell and L. Simoncini Ed., Springer-Verlag, 1992, pp. 113-135.

[16] A. T. Tai, A. Avizienis and J. F. Meyer, "Performability Enhancement of Fault-Tolerant Software," IEEE TR, Sp. Issue on Fault Tolerant Software, Vol. R-42, pp. 227-237, 1993.