Optimizing the Reliability of Component-Based n-Version Approaches
K.E. Grosspietsch
Fraunhofer Institute of Autonomous Intelligent Systems
D- 53754 St. Augustin, Germany

Abstract
In this paper a component-based approach for implementing n-version software systems is presented. For each of the system modules, a set of diversely developed module candidates is considered. It is demonstrated that in general such a modular construction offers a potential to considerably increase the reliability of the n-version system. Moreover, we describe an optimization approach to further optimize system reliability by selecting the most adequate candidates while at the same time observing given system constraints. A heuristic search algorithm to find optimal system configurations is sketched. Several modifications of this algorithm are discussed. Simulation results concerning the outcome of these different approaches are presented.

1 Introduction
Our society more and more relies on the dependable execution of complex concurrent program systems, where a large number of software components has to communicate and co-operate. Software design technology has made some considerable progress in the last decades, but this has been compensated on the other hand by the strongly growing complexity of systems. Thus, methods of fault tolerance are continuously being discussed to overcome these difficulties. The standard strategy for software to realize fault tolerance for software is design diversity. Here two main methods have been proposed: n-version programming [1] and recovery blocks. Apart from the discussion of the stochastic independence of faults within different versions, the main tradeoff of the methods is the resulting dramatic increase of the costs for software production.

On the other hand, today also more and more the construction of software systems from libraries of existing standardized components is being discussed. Here two main reasons might be seen:

1) The reduction of development costs;

2) the possibility to test the reusable software more intensively.

All these approaches - for cost reduction as well as for gain in dependability - are characterized by the fact that a larger target system is created by selecting and aggregating components from a set of candidate building blocks. In all these cases it would be very helpful, if not only the costs, but also the dependability properties of the system to be built could already be estimated from those ones of the given components [2,3,4]. So a dependability analysis for such cases should also be able to take such factors into account.

On the other hand, when several system parameters have to be observed, as e.g. system reliability, costs, or upper bounds for execution time, with rising system complexity the number of possible system configurations and constraint combinations is exponentially rising. Thus an exhaustive comparison among the different combinations of candidate modules might quickly exceed the available computing time. This holds especially if for each system module not only a single candidate, but several out of the given candidate set have to be selected, as it is necessary in the case of n-version systems.

We propose here to systematically utilize, for future developments, the combined application of standard software components, and design diversity. I.e. different candidates for versions of a software component should be seen to be gained:
1) from different libraries;

2) by parameterizing or partially reprogramming existing modules;

3) by combining existing software components with newly programmed software to form an n-version implementation of that system part.

In addition, for the organization of the n-version system we propose the following principles:
1) It should be possible to define, for different sections of the system, according to their varying degree of criticality, different degrees of redundancy (no redundancy, 2 versions for error detection, 3 versions for simple fault tolerance, etc.); the voting procedure thus has to adapt then to a varying number of versions to be considered.

2) The selection of the needed versions should be carried out as systematically as possible.
To reach this goal, we have developed an integrated approach to derive the optimal dependability structure of n-version software systems while considering basic information on reliability, on the cost of modules, and on the timing and resource requirements of task execution.

The systematic consideration of constraints within the optimization process has one important consequence: For the constraint checks always complete system configurations have to be created. i.e. it is not possible to carry out the optimizing selection of candidates for the versions of each system part separately. In this situation, the use of systematic, exhaustive comparison algorithm would mean that all potential system configurations have to be tentatively generated, checked for the fulfillment of the side conditions and processed to compute the corresponding overall system reliability. This usually would cause a computing complexity that is untractable even for the most modern high speed computers. Therefore, as central part of our reliability optimization procedure, we have developed an adaptive random search algorithm.

2 Basic System Structure

In most applications, not all of the parts of the software system to be implemented, have the same critical behavior with respect to faults. So, for some of them, two versions (for error detection) might be sufficient, whereas for other more critical parts three versions (as the simplest fault masking scheme) or higher n- (n > 3) version redundancy should be applied. The varying degree of redundancy of course implies that the voting procedure has to be tailored to a varying number of versions. In contrary, however, to hardware systems, where the voter is usually realized by an extra hardware circuit, the voter for software systems is a quite simple software routine, which easily can be tailored, by using the number of versions to be voted as a procedure parameter.

Let us consider the following model of a software system to be built: During its application, the software system has to solve a number of I tasks; to do so, we have to assign to each of the tasks a certain software module i (i=1,...,I). Let us assume that when using n-version programming, we have Ki candidates for realizing module i, and ni (n=1,...,Ki) of these candidates are selected as module versions (see Fig. 1). The selected ni versions of module i together form the module stage i of the system.

Which of the candidates have been selected, is denoted by the system configuration array X: This array is an I x n J x K array where J is the maximum of the values ni (i=1,...,I) and K is the maximum of the values Ki (i=1,...,I). If candidate k (k=1,...,Ki) is selected to implement version j of module k, then X[i,j,k] is set to the value 1, otherwise this component has the value 0.

3 Reliability of the Component-Based System

Let us introduce the following notations for a reliability model:

\[ R_j \] Estimated reliability of module i (i=1,...,I) of version j (j=1,...,ni);
\[ R_{mi} \] Estimated reliability of module stage i;
\[ R_c \] Estimated reliability of the entire software.

The use of modular components to build the n-version system has a very interesting implication. The interfaces between two subsequent module stages i and i+1, i.e. between the versions j (j=1,...,ni) of subsequent modules and i and i+1 (i=1,...,I-1) here offer another possibility: Whereas in «monolithic» versions just the outputs to the system environment are protected by voting, here we can vote also over the data transferred between these stages. This voting can be utilized for a state restoration of the system which in the case that, due to a fault, one version j of module i is deviating from the majority behavior, automatically corrects the erroneous system state. By simply taking the voted data, i.e. those ones which the majority of the versions of module i has produced, instead of that one of the fault-affected version, the potential effect of the fault for the subsequent version j of stage i+1 is removed.

In the following we shall demonstrate that the reliability of such a modular n-version system using the described state restoration method, is larger than that one of a comparable conventional n-version system. We shall show that here for the most important practical case, a 3-version system with versions j (j=1,2,3) having the reliabilities R1j, R2j, and R3j. The derivation is carried out by induction: Let us first consider the case of two module stages, i.e. I=2. Thus considering each of the versions as composed of 2 modules with reliabilities Rij, R3j, (j=1,...,3), their reliability can be also expressed as R1 = R11 * R12, R2 = R21 * R22 and R3 = R31 * R32.

The system reliability of a conventional 3-version system is given by the well-known formula

\[ R_c = R_{\text{voter}} * (R_1 * R_2 * R_3 + R_1 * R_2 * (1-R_3) + R_1 * R_3 * (1-R_2) + R_2 * R_3 * (1-R_1)) \]  (1)

This formula just states that the system works properly as long as the voter is fault-free, and not more than one version is failing (i.e. either all versions are fault-free or at least two of them).

The voter procedure performs a very simple task of counting the numbers of different values found and looking whether one of the values has a majority; so this procedure needs only a small number of lines of codes. As a consequence, here it can be assumed that by rigorous software verification/testing techniques the goal is reached that this small part of the software...
The term $1-R_{j} = R_j^*(1-R_{j}*)$ can also be expressed by the expression

$$1-R_j = R_{1j}*(1-R_{2j}) + R_{2j}*(1-R_{1j}) + (1-R_{1j})*(1-R_{2j}); \quad (2)$$

this expression just denotes the probability that version $j$ is faulty because either module 1 of it is fault-free, whereas module 2 is faulty; or vice versa, or both modules are faulty.

Substituting this into eq. 1, we obtain the following sum containing 10 terms:

$$R_c = \sum_{i=1}^{10} R_{i1}R_{i2}R_{i3} \quad (3)$$

Alternatively, in the case of a modular 3-version system, for both of the module stages 1 and 2 their reliability can be expressed, in terms of the module reliabilities, by formulas analogous to (1):

$$R_{m1} = \prod_{i=1}^{3} R_{i1} \quad (4)$$

The reliability of the entire modular system consisting of 2 module stages is then given by the product of these two reliabilities:

$$R_{mod,2} = R_{m1} \cdot R_{m2} \quad (5)$$

By multiplying out these two expressions, we get a sum of 16 terms, 10 of which are identical with the terms given in eq. 3. The remaining 6 terms are

$$C = \sum_{i=1}^{6} C_{i} \quad (6)$$

This means that

$$R_{mod,2} = R_{m1} \cdot R_{m2} = R_c + C, \quad \text{hence} \quad R_{mod,2} > R_c \quad (7)$$

The six terms of sum $C$ just represent those cases that after a fault has occurred within module 1 of one of the 3 versions (and its effect removed again by state restoration!), a second fault might occur in the module 2 of one of the other two versions. These are cases which in a conventional 3-version system cannot be tolerated.

After for the case of $I=2$ modules our assumption thus has been proven, the transition from $I$ to $I+1$ module stages is simple. Just consider the second module stage $m_2$ to be split up into two stages $m_2'$ and $m_2''$ with voting and internal state restoration at the interface between them. Then a consideration analogous to that one above leads to

$$R_{m2'} \cdot R_{m2''} > R_{m2} \quad (8)$$

So,

$$R_{mod,I+1} = R_{mod,I} \cdot R_{m1} \cdot R_{m2'} \cdot R_{m2''} > R_{mod,2} = R_{mod,I} \quad (9)$$

This completes the proof.

The derived results can also be made evident by lucid qualitative arguments: In the 3-version system without state restoration, the system fails if somewhere (i.e. at any point) in two of the versions a fault has occurred. In the modular system with state restoration at module stage level, the system is failing only if faults occur in different versions of the same module stage; this is much less probable.

Fig. 2 shows the effect of using the modular 3-version approach for the example of 7 reliability critical components of a satellite control system [8]. It can be seen that by the modular approach, a considerably higher system reliability is gained.

Generalization from the case of $J=3$ (3-version system) to arbitrary $J$ is possible in similar ways, but shall not be considered here due to the limited page volume of this paper.

### 4 Reliability Optimization

#### 4.1 Constraints for System Optimization

Of course, system reliability does not only depend on the choice of the overall system organization, but also on the reliability of the individual modules who implement the (redundant or non-redundant) system. Under the assumption that these individual reliabilities of the building-blocks are known or can be estimated, we can see to derive an optimal system configuration which maximizes overall system reliability:

$$R = \max \prod_{i=1}^{J} R_{im} \quad (10)$$

Usually, however, there are additional constraints under which the optimization has to be carried out. As examples, we shall consider here two constraints with regard to cost and execution time bounds: In terms of the configuration vector, the cost constraint which guarantees that total expenditures will not exceed an upper bound $B$ can be formulated as follows:
the software system is activated to execute the (consider \( \frac{5!}{(3! \ 2!)} \)) 64 different system from 5 different candidate modules, we would have to tripllicated, thereby selecting each of the module versions of task i, realized by candidate module k for module stage i.

As another example of a constraint, let us consider a maximum bound T for the execution time. The progress of execution of the application process can be formulated by means of an evolution time vector (ETV) of the task system which is given by:

\[
    t = \{t_1, .., t_i, .., t_I\},
\]

where \( t_i \) is the point of time when a component of the software system is activated to execute the \( i \)-th (\( i=1, ..., I \)) task. The durations of task executions is formally described by a duration vector; its component \( d_k \) corresponds to the duration of executing task \( i \) by means of module candidate \( k \) (\( k=1, ..., K_i \)).

For our case let us consider the following constraints to the maximal time duration:

\[
    \text{max} \ (t_i + d_k) - \text{min} \ t_i \leq T, \quad i = 1, ..., I
\]

This relation simply states that for all tasks \( i \), the completion of the task does not exceed the required upper bound \( T \) for the execution time of the entire software process (see Fig. 3).

Additional constraints might be the availability of system resources as e.g. I/O channels etc.; we shall not consider this here in detail. There is one important consequence of having to consider constraints: The optimal selection of modules for system tasks cannot be performed independently, task by task. Instead, always complete potential system configuration have to be generated and checked whether they fulfill the given system constraints. In the next section we shall present an approach to cope with this situation.

### 4.2 Basic Approach

To derive an optimal reliability solution by means of a systematic, exhaustive comparison algorithm would mean that all potential system configurations have to be tentatively generated, checked for the fulfilment of the constraints and processed to compute the corresponding overall system reliability. This usually would cause a computing complexity that is untractable even for the most modern high speed computers; If, e.g., we consider a system consisting 64 modules, all of which are to be tripllicated, thereby selecting each of the module versions from 5 different candidate modules, we would have to consider \( \frac{5!}{(3! \ 2!)} \) \( 64 \) = 10 \( 64 \) different system configurations! Assuming e.g. 1 nsec for processing each system configuration (of course, a value by far too optimistic!), the resulting 10 \( 55 \) sec of needed computation time would exceed the estimated age of the universe of about 10 \( 17 \) sec by many orders of magnitude! Therefore, here only stochastic search methods appear possible to provide, in a heuristic way, an optimal solution.

In the field of pseudo-Boolean optimization theory, the so-called method of varied probabilities (MVP) has been developed to solve complicated problems, especially ones with a large dimensionality [5,6,7]. The method of varied probabilities (MVP) presents a family of heuristic algorithms based on one common scheme. In order to find an extremal solution of a pseudo-Boolean optimization problem, a probability vector of dimensionality of the sought solution vector is formed. Each component of the probability vector presents a probability of assigning a one value to the correspondent component of a Boolean system configuration vector. In the terms of developing an N-Version system (NVS) structure, it looks like a probability to include a version candidate into the system structure.

The initial values of the probability vector components describe a situation where every module candidate has the equal probability to be included into the system structure. Then, at a computational phase, random decisions are generated according to the probability distribution specified by means of the probability vector. For each of the generated samples, the corresponding objective function which denotes the “appropriateness” of this system configuration, is calculated. Then, according to the outcome of these values, those ones of the probability vector components are updated, thus changing the profile of the probability distribution. The ways of changing these values define the individual algorithms of the MVP scheme. The common approach for updating a probability vector can be characterized by the rule: the better result received with a one-valued binary vector component the bigger probability value is assigned to the corresponding position of the probability vector.

As one basic example of implementing an MVP algorithm in detail, in the next subsection we shall discuss the so-called adaptive random search algorithm. Then in section 4.3 we shall discuss potential modifications of this solution.

### 4.3. The Adaptive Random Search Algorithm

The adaptive random search algorithm (ARSA) is one of the possible concretisations of the MVP scheme; here the 7 steps of the search procedure have the following form (as an example of the objective function, here we consider the probability of success as introduced in section 1):

1. The initial values of the probabilities vector components \( V^0 \) are given. In the simplest case the
ARSA starts with equiprobable distribution of the unit variables of the probabilities array.

2. A number of q system configuration arrays $X_s (s=1,...,q)$ are selected. This selection is an independent random sampling (according to probability distribution $P'$, $f=0,1,...,Q$ (Q being the number of loop repetitions of the algorithm).

3. The values of the reliability corresponding to the q system configurations $X_s (s=1,...,q)$ are computed.

4. The values $R_{f_{\text{min}}}^{'}$ and $R_{f_{\text{max}}}^{'}$ are defined by the conditions

\[
R_{f_{\text{min}}}^{'} = \min_{s=1,...,q} R(X_s),
\]

and

\[
R_{f_{\text{max}}}^{'} = \max_{s=1,...,q} R(X_s).
\]

The two arrays $X_{f_{\text{min}}}^{'}$ (corresponding to the worst value $R_{f_{\text{min}}}^{'}$ ) and $X_{f_{\text{max}}}^{'}$ (corresponding to the best value $R_{f_{\text{max}}}^{'}$ ) are memorized.

5. Adaptation in this algorithm is a modification of the vector $V$ according to the information which for the preceding algorithm steps was obtained:

\[
V := V + Y*X_{f_{\text{max}}}^{'} - Y*X_{f_{\text{min}}}^{'}
\]

where $Y = 1/(q*L)$ (L being a scaling factor);

i.e. all the components of $V$ which in their index position within $V$ coincide with a "1" in the the array $X_{f_{\text{max}}}^{'}$ are increased by the value $Y$, all components which coincide with a "1" in the array $X_{f_{\text{min}}}^{'}$ are decreased by the value $Y$.

6. Steps 2-5 are repeated Q times.

7. A solution of the optimization problem is the vector $X'$ defined by the condition:

\[
R(X') = \max_{f=1,...,Q} R_{f_{\text{max}}}^{'}.
\]

This optimization method has already successfully been in practice as a decision support tool for a satellite control system [8]. The system consisted of 64 modules; 12 of them were selected to be implemented in a redundant way. 5 of them were just implemented in 2 versions, for error detection. The 7 remaining ones were to be realized in 3 versions, for fault tolerance.

4.4 Modifications of the ARSA Algorithm

Initially, ARSA has been developed for the problem of pattern recognition to select an informative subsystem of attributes. The main disadvantage of this algorithm is a potential problem of updating values of a probability vector components. Namely, in some cases it is possible to get the values of intermediate solutions which do not let the probability vector components to be changed any more. I.e. the algorithmic solution does not improve any more. To correct the defect, a modification of ARSA has been developed (Modified ARSA ver.1). The statistical data of applying the modified version of ARSA shows a better algorithmic behaviour when solving problems of developing an NVS structure.

Moreover, ARSA does not provide a technique of avoiding zero-solutions (i.e. configurations where for one or more of the tasks no module candidate is assigned) when solving the problem of designing NVS structure. To protect an algorithm against spending both computational and time resources for calculating the objective function values in the points of this kind, a particular technique of generating random non-zero solutions has been developed. This technique is utilized in the MVP based algorithm named NVS MVP (denoting the restricted application field for which the algorithm is used).

The objective function of the presented optimization problem has several specific features which can assist to reduce a search domain, thus allowing to decrease the searching time. So then, the objective function as a function of the whole system reliability represents the product of reliabilities of separate software modules. Consequently, when a reliability of any of the modules is equal to zero the overall system reliability turns into zero value also. Physically, it represents a case when there are no versions chosen for (at least) some of the software modules. The configuration vector components corresponding to such software modules will be assigned zeroes as well. Obviously, it is necessary to avoid computing the objective function in such the points.

Making use of both of the mentioned enhancements considerably increased the efficiency of applying the MVP based algorithms to the problem of NVS structure development. The statistical results presented in the final part of the paper show it. Different algorithms have been tested on the same optimization problem with the same quantity of objective function calls.

Another random algorithm is the algorithm of random search of boundary points [9]. It is based on the proven fact that a solution of the stated optimization problem is a so-called boundary point, i.e in terms of binary space topology, a point neighbouring to the set of infeasible solutions. Such a point describes a system configuration which cannot be extended by including any additional version module, without violating the given optimization constraints. The algorithm of random search of boundary points generates multiple boundary solutions and compares their corresponding objective functions values.

In the following, we shortly sketch the algorithm of generating boundary points; different boundary points can be reached using this algorithm when different combinations of ways to choose oindex b, at the second step of the algorithm.
1. The initializing step: \( i = 0 \).
2. Determine a boundary point \( X_{b_i} \) (b – as an index means “boundary”).
3. Calculate the objective function value \( F_i = F(X_{b_i}) \).
4. If the stopping condition is satisfied go to p. 5, otherwise \( i = i + 1 \) and go to p. 2.
5. The solution is \( F^* = \max_i F_i \).

Separate variations of the algorithm of boundary points search may differ from each other in a stopping condition and in ways of reaching boundary when generating boundary points. For the optimization problems of high complexity it is more rational to use stochastic version of the algorithm when boundary points are reached in a random way and this process is executed repeatedly.

Statistical data [9] show among the different ARSA modifications, the boundary search algorithm produces the best values, which also show little fluctuations.

5 Conclusion

In this paper we have discussed a component-based approach for implementing n-version software systems. For each of the system modules, a set of diversely developed module candidates is considered. It is demonstrated that in general such a modular construction offers a potential to considerably increase the reliability of the n-version system. Moreover, we present an optimization approach to further optimize system reliability by selecting the most adequate candidates while at the same time observing given system constraints. A heuristic search algorithm to find optimal system configurations is described. Several modifications of this algorithm are discussed. Simulation results concerning the outcome of these different approaches have been presented.

6 References

Fig. 1: Structure of a modular n-version system
Mij version j of module i
Fig. 2: Cumulated reliability of 7 components of a software system for a satellite control application
a: non-redundant system configuration; b: conventional 3-version system configuration; c: modular 3-version system

Fig. 3 Time constraint for the tasks i(i=1,...,I)