# VITAL CODED MICROPROCESSOR PRINCIPLES AND APPLICATION FOR VARIOUS TRANSIT SYSTEMS

**P. Forin**

*Research and Technology Department, Matra Transport, 56 rue Barbès, 92120 Montrouge, France*

**Abstract :** In this paper we consider an application of redundancy codes to the problem of vital data processing. In a first step, we introduce the random codes in application of digital data links and we demonstrate the efficiency of this probalistic model to obtain the highest security levels.Thereafter we describe how these concepts are used to design standard coded operators, to process and compute vital data with non-redundant hardware and software. This Vital Coded Processor is now in use on several automatic and semi-automatic guideway transit systems.

**Keywords :** Coding, random processes, error detection codes, computer organization, railways, safety

## 1. PRINCIPLES OF PROBABILISTIC SAFE DESIGN

The use of redundant codes for the detection of errors occuring in data processing has the same bases as those used for the protection of digital links.

So we are going to introduce the principle of the Coded Processor in the more simple case of a data transmission protection.

### 1.1. Basic hypothesis

In order to replace VITAL data transmission systems used until now and designed according to fail-safe rules by digital coded links, we need to determine the acceptable residual error rate (i.e. not detected). In order to have a wide margin in respect to the level of safety obtained by conventional techniques, we had initially set up at $10^{-12}$ the maximum non detection probability of an error.

Such a goal cannot be obtained by the usual coding methods which requires a probabilistic error model for the choice of the error detection code : if the real behaviour of the transmission channel diverges from the expected behaviour, the error detection performances of the code are ignored.

We have therefore abandonned the idea of defining an error model of the channel, that induces a new difficulty : upon what random process relies the error detection probability ?

Without an error model the possible errors remain just as random or deterministic,but are unknown : the random process in which we are interested has to be seeked in the protection code itself.

### 1.2. Random codes

The construction of a code consists in selecting, among the N messages that may be transmitted, the $M \leq N$ different words that constitute this code.

If those M messages are obtained by random sampling among the N possible words, the probability that any error (with random or deterministic origin) transforms a message (belonging or not to the code) into a message belonging to the code, is equal to the ratio M/N of the number of words of the code to the number of possible words.

Let $P_{obj}$ be the level of safety to obtain and k the number of check bits of the message.

We have
$M/N = 2^{-k}$
and we must get
$M/N \leq P_{obj}$
hence $k \geq \log_2 P_{obj}$
which gives : $k \geq 40$ for $P_{obj} = 10^{-12}$

This result allows us to link directly the security level from the system and the number of check bits, independantly of the number of messages to be transmitted and of the transmission channel link.

Practically, in other respects, it is not necessary to sample in a random way all the code words to obtain this result. It is enough to use a code construction process following the statistic properties of random sampling.

## 2. CONSTRUCTION OF THE CODED PROCESSOR

We have established that it is possible to construct redundant codes which allows us to obtain any safety goal in regard of errors disturbing a data transmission, whatever the errors origin (random noise, circuit failure...). We are now going to apply the same principles to detect the errors occuring into a data processing inside a microprocessor.

Two new constraint types therefore appear.

### 2.1. Fonctional constraints

Carrying out a coded addition means that if two coded operands are presented to the addition operator, the latter has to produce the coded results without the need to decode the variable during the operation. If we want to apply a random code to operation, it is necessary to provide a 2 dimensions transcription array addressed by each operand : the array capacity will be then $N^2$ elements whose $M^2$ are the results of coded additions.

Such a construction being not practically feasible, we had to find a coding process allowing an easy carrying out of the addition operator while preserving the properties of random codes.

It is the same for all the operators that are necessary to carry out complex fonctions : multiplication, conditional branches, logical operators...

### 2.2. Safety requirements

The whole set of possible failures able to disturb a microprocessor and its associated circuits may induce at the level of processed data only 3 types of errors :

- Operation errors

The result of processing operands with expected operator is false. This type of error, very similar to transmission errors, does not introduce new constraints upon the code design.

- Operator errors

The processor uses the good operands but with an unexpected operator. For instance, if an addition operator is replaced by a multiplication operator, the result is false even if the multiplication is good. The detection of this type of error, that does not exist in a data transmission channel, needs to use a code specific to each operator.

- Operand errors

This error type, which consists in applying the right operation to unexpected operands, may take two aspects :

- an address error that is equivalent to the replacement of a variable by another. This type of error occurs also in a transmission system in case of crosstalk between separate channels : we may then receive a message sent to another receiver.
- a memory «non-refreshed» error which is the substitution of a current value by the value of the variable computed in a previous cycle

As for the detection of operator errors, different codes have to fit with different operands, in order to detect those failures.

### 2.3. Construction of the code

First step

We have seen that functional constraints require us to have operators transforming directly coded operands into coded results. In order to make this operation as easy as possible with integer numbers, we retained codes of the following form :

$$X = A.x$$

where X is a coded operand
x is the uncoded value of this operand
A is randomly selected number, such as $1/A \geq P_{obj}$

The set of words of the code is therefore constituted by the multiples of A, that allows a direct carrying out of addition and multiplication operators.

$$Z = X + Y \Longleftrightarrow Z = A.x + A.y \Longrightarrow Z = A\,(x + y) = A.z$$

With this form, the code allows the detection of operation errors with the probability P=1-1/A, but does not detect neither operand nor operator errors because all variables are multiples of A : we are therefore going to introduce a «signature» specific to each operand.

Second step

The introduction of this static signature consists only in the addition of a known constant to each operand, those constraints being choosed randomly among numbers lower to A :

$$X = A.x + B_x$$

With this code, the addition of two variables X and Y gives:

$$Z = X + Y = A\,(x + y) + (B_x + B_y)$$
$$= A.z + B_z \text{ if } B_z = B_x + B_y < A$$

The signature of the result, that is the remainder of the division of Z by A depends concurrently upon :

- signatures of operands ($B_x$ and $B_y$),
- and of the good operation execution : the non detection probability of any combination of these 3 errors is still 1/A.

We have seen that the signatures $B_x$ and $B_y$ of operands X and Y are chosen randomly, but this is not the case of $B_z$ which is the sum of the two previous signatures.

We could add to Z a constant $C_z$ choosen randomly in order to enforce this hypothesis, but this supplementary operation does not change the error detection probability. It is therefore enough to choose randomly the signatures of input values of the system, and to let the signature to evoluate at each addition of the program sequence.

This result is also true for the multiplication if A has been chosen a prime number.

Third step

If the introduction of the static signatures $B_x$ allows the code to detect addressing errors, it does not solve the problems created by the use of a same variable during successive cycles, unless the whole set of signatures is changed at each cycle.

A more simple solution consists in adding to each coded operand a dynamic signature of the logic time, that we called the date D, identical for all the variables and whose value is modified at each iteration cycle.

Finally, a variable x is coded under the form :

$X = A.x + B_x + D$

In order to give all operands the same dynamic signature D, this date has to be known by the processor and explicitely used by the coded operators.

For instance, for the addition operator, we have :

$Z = add (X, Y)$
$= X + Y - D$
$= A.z + B_z + D$

This date is transparent for all operators and will not be mentioned in further descriptions of this presentation.

Fourth step

In order to simplify the program tests, we replaced the code form

$X = A.x + B_x$

by a representation with separation between information and check bits.

Let k any number such as
$2^k > A$ and
$r_k = 2^k \text{ MOD } A$
$r_{kx} = 2^k.x \text{ MOD } A.$

The quantity $2^k.x - r_{kx}$ is a multiple of A.

We may therefore, without changing the properties of the code, represent coded variables under the form :

$X = 2^k x - r_{kx} + B_x + D$

In this new representation, the check part occupies the k low significant bits and the information occupies the most significant bits, which allows an effective separation in two fields :

$X = (x, C_x)$ with $C_x = -r_{kx} + B_x + D$

## 2.4. Main coded operators

Addition

This is the more simple operator, because the sum of two coded numbers provides directly the coded result

Multiplication

With two operands X and Y, this operator provides a result $Z = Mult (X, Y)$ whose signature is $B_z = K . B_x B_y$, that is therefore different of the signature $B_x + B_y$ provided by the addition operator.

Comparison and conditional branch

Let the following algorithm :

IF $T \geq 0$

Then $Z := X + Y$

Else $Z := W$

ENDIF

At the end of the test, the signature of Z has to be affected by a branch error but, if no error occurs, the signature has to be unique after the convergence between the 2 conditional branches : this condition is required to be able to check the signatures of the final results in relation with predetermined values.

In order to obtain a signature of the comparison $T \geq 0$, it is enough to compute the unsigned division of the coded variable T (whose signature is $B_t$) by A.
The remainder R is equal to :

- IF $T \geq 0$ then
  $R1 = B_t$
- IF $T < 0$ then the internal machine representation is $2^n + T$, if n is the number of bits of the machine word.

  The remainder of the unsigned division of T by A is therefore:
  $R2 = B_t + r_n$ with $r_n = 2^n \text{ MOD } A$

In this way, we have a «marker» that depends of both the test variable (by the term $B_t$) and of the result of the comparison (term $r_n$ if $T < 0$) : we have then realized a comparison operator.

In order to avoid that the difference between the two signatures be the same for all tests ($R_2 - R_1 = r_n$), we compute the signature S of the comparison by applying to the R mark a function $F_i$ different for each test : we obtain then a signature $S_i$ that takes the values $S_1$ or $S_2$, we have just to add to the variable in order to obtain a signature depending upon the result of the comparison.

While adding a precomputed constant $C_{iz}$ to the variable Z in one branch of the test, we obtain a final signature $B_z$ independant of the selected branch if there was no branch error.

Full branch structure algorithm

| Program | | Signatures of the variable Z |
|---|---|---|
| R := | T unsignedMOD A | |
| $S_i$ := | $F_i$ (R) | |
| IF | T ≥ 0 | |
| then | Z := X + Y | $B_z = B_x + B_y$ |
| else | Z := X | $B_z = B_w$ |
| | Z := Z + $C_{iz}$ | $B_z = B_w + C_{iz}$ |
| ENDIF | | |
| S := Z + $S_i$ | | $B_z = B_x + B_y + S_{i1}$ |
| | | $= B_w + S_{i2} + C_{iz}$ |
| | | with $C_{iz} = B_x + B_y - B_w + S_{i1} - S_{i2}$ |
| | | ( ) |
| | | value precomputed and stored |

Loop structures

The direct use of the coded conditional branch to carry out the exit test of a loop needs a compensation set (one constant $C_i$ for each variable modified) different at each cycle.

The introduction of the date D (dynamic signature), incremented by a constant dD (different for each loop structure) at each loop cycle allows us to use only one compensation constant for each variable modified in the body loop :

Let $D_0$ the value of the date D at the entrance of the loop. The full signature of a variable X is therefore $S_x = B_x + D_0$. After execution of the loop body, the signature is now :

$S'_x = B'_x + D_0$

If we add to this new signature the constant $C_x = B_x - B'_x + dD$, we obtain a new signature $S_{1x} = B_x + D_0 + dD = B_x + D$ of the same form as the initial signature.

Then, after the loop exit, remains to substract to the variable X the difference between the initial date and final date (n.dD if we did n iterations in the loop) in order to obtain the final signature :

$S'_x = B'_x + D_0$, independant of the loop cycles number (in absence of errors).

The other operators

Using the 3 basic operators : addition, multiplication and conditionnal branch, we have also defined the following operators :

- integer numbers operators
  - substraction
  - trunc of p bits
- boolean operators
  - AND, OR, NOT, Exclusive OR
  - IF X = TRUE THEN ——— ELSE ——— END

and so the multidimensional array structures using coded index.

## 3. ARCHITECTURE OF THE VITAL CODED PROCESSOR

### 3.1. Hardware organization

Most vital coded processor are provided by traditional fail-safe parts. So it is necessary to convert all these data into coded streams at the microprocessor input. This operation is carried out by input fail-safe coders that provide a specific signature for each «high level» input.

In the same way, most vital coded outputs have to be converted to fail-safe signals. This process is carried out in 2 steps :

- each output is inversed (with a fail-safe function) and is converted to a coded data stream and permanently checked by the microprocessor.
  If the value checked does not match with the internal value of the microprocessor, the global signature streams given by the microprocessor will be wrong.
- the global signature stream generated by the microprocessor is processed by a fail-safe dynamic controller (by comparison with a predefined sequence) that provides the energy to all vital outputs : any error disables this energy supply and all vital outputs are passivated (at the restricted state).

As any iterative structure of the coded vital processor, the main program loop produces a dynamic signature (level 0 date) whose normal evolution ensure the correct real-time operation of all program tasks.This date is also checked by the dynamic controller from a fail-safe clock generator.

### 3.2. Software production

The coded vital programs obey standard vital software production cycles up to the source language generation (Modula 2 now). The source file is then :

- compiled for the generation of the machine code
- analysed in order to determine all signature and compensation constant tables that will be used by the program.

At this level, the safety is obtained by an association of both coding and heterogeneous redundancy between the compiler and the software tool generating the signatures.

After this phase, the safety is fully ensured by the coding and the program is processed like any program : link, loading.

All errors generated into these two steps will have the same effects as the target microprocessor failures : they will be detected by the code as soon as they disturb the program execution.

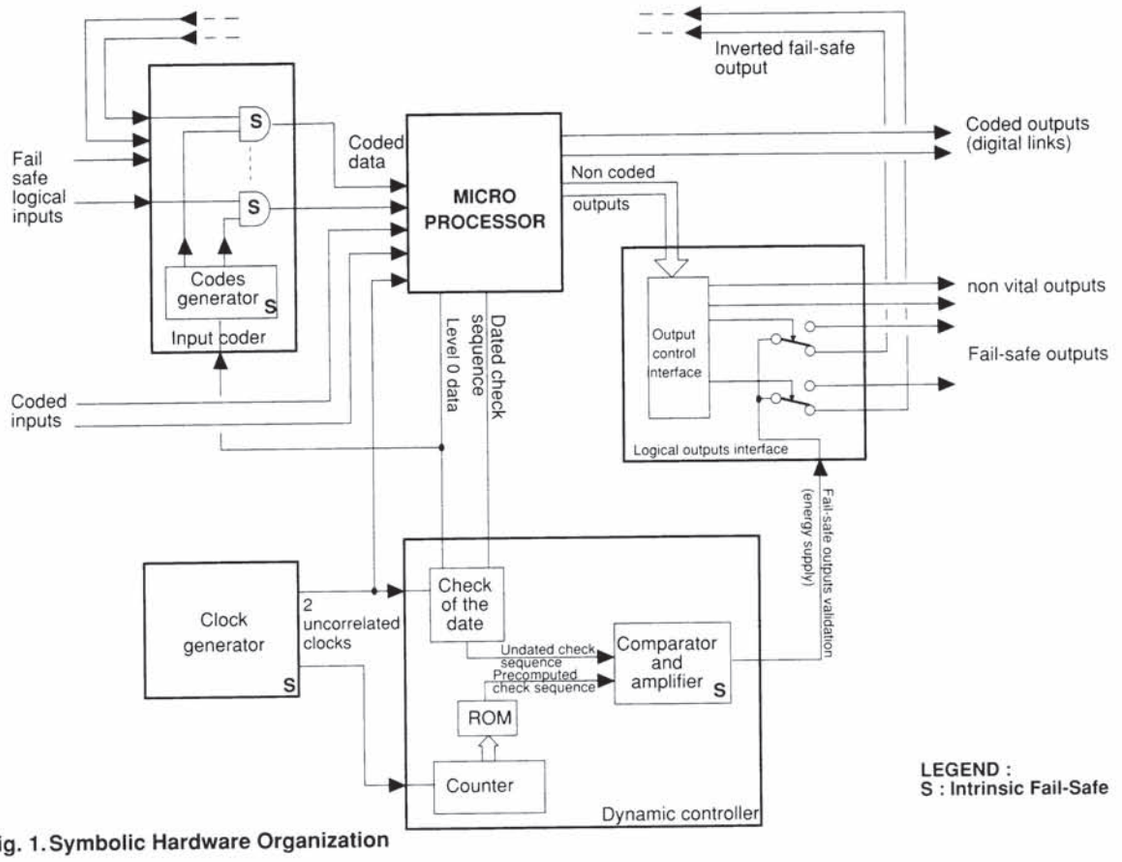


**Fig. 1. Symbolic Hardware Organization**

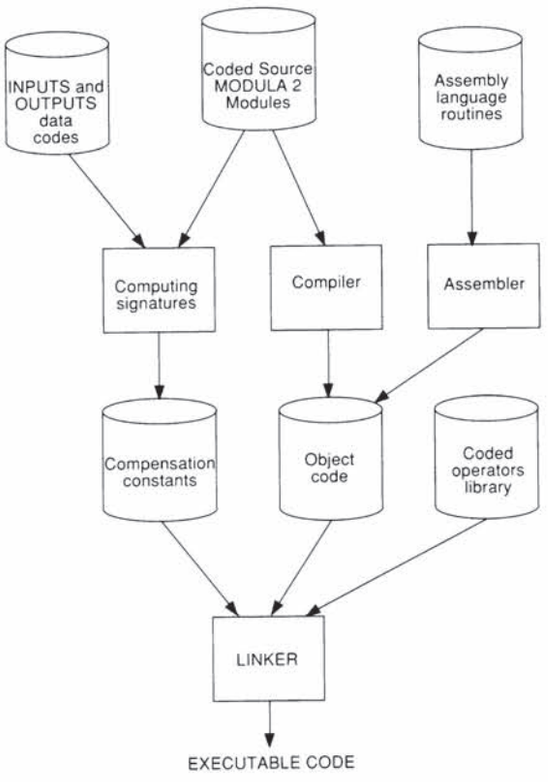


**Fig. 2. Software Generation Process**

## 4. SOFTWARE ERRORS AUTOMATIC DETECTION

As the code brings a full protection only from the compilation phase, two important categories of analysis and programmation errors are also covered by the code : the errors concerning the data structures handling and the real-time organization of the program tasks. They are in other respects, the most difficult (or even impossible) to detect by tests. Rather than a formal description, let us consider 3 simple examples :

- Overflow

  The detection of this error lays on the same principle than the comparison between two integers : for n bit words, the static signature of the considered variable will be altered of 2n MOD A in case of overflow of 1 bit, during an addition for example.

- Index array out of range

  In this case, the system acts in the same way as an address error: if the error happens at the reading of an array element, the signature obtained will be the residue modulo A of the addressed zone ; if the error happens at the writing of an array element, we may overwrite an other coded variable and the error will be detected at the use of this variable (in the opposite case the error has no manifestation), we may be lucky enough to avoid any program disturbance.

- Errors of synchronization between tasks

  We have seen that the global program signature provided to the dynamic controller depends upon the signature of any operand used, for the static signature as well as the dynamic signature (the date) : any failure in the order (or duration) of tasks activation disturbing the results will also have an effect on the signatures.

  This point has a particularly important consequence : since the code detects any error in the real time organization of the task, it is possible to use any of the shelf real-time kernel without the need to validate it.

## 5. CONCLUSION

The Vital Coded Processor has been originally designed for the protection of trains on the PARIS RER A line in the SACEM system that was put into operation late 1988.

After this first application organized around a MC 68 000 processor, we have choosed this technology for the vital processing of the fully automatic LYON metro D line (MAGGALY system).

In order to face new requirements in terms of computation power, we have upgraded the architecture by use of a MC 68020 CPU board : in accordance with our initial objective (errors detection probability independant of the hardware), no supplementary safety validation has been needed.

This new version is the heart of all our new transit protection systems in development, particularly the CHICAGO AIRPORT VAL application that uses the coded vital processor for the carrying out of logic interlocking and signalling.

For further applications, we focus our efforts to run time reduction, by the carrying out of a coprocessor able to process directly coded operations in the same way a floating point coprocessor may be associated to a standard processor.