FORWARD ERROR RECOVERY: TOWARD A PREVENTIVE METHODOLOGY[1]

A. JAOUA[2], K. ZEROUAL[3], N. BRIÈRE[3], P. TREMBLAY[2] and R.O. ZAIANE[2]

ABSTRACT

In this paper we propose a new method for the construction of fault tolerant programs: the "preventive" method, which is based on forward error recovery. With this method, the execution of the program is controlled relatively to its specification, so that we never try to detect the presence of any error nor to find the faults which cause the error, but we change the current (correct or erroneous) state of the program into a new correct state. Tested on very contaminated programs, this method has given satisfactory results with a better run time than another forward error recovery methodology: hybrid validation [Mili, 85]. It also appears to be more efficient than the executable assertions method [Jaoua and Mili, 90].

Keywords: Forward Error Recovery, Fault Tolerance, Reliability, Relational Specification, Difunctional Relations.

RÉSUMÉ

Dans cet article, nous proposons une nouvelle méthodologie de construction de programmes tolérants aux fautes: la "méthode préventive", qui est basée sur le recouvrement avant des erreurs. Avec cette méthode, l'exécution du programme est contrôlée relativement à sa spécification, de telle sorte que nous n'essayons jamais de détecter la présence d'une erreur, ni de trouver les fautes qui ont provoqué ces erreurs, mais nous changeons l'état courant du programme (correct ou erroné) en un nouvel état correct. Testée sur des programmes très contaminés, cette méthode a donné des résultats satisfaisants avec un meilleur temps d'exécution qu'une autre méthodologie connue de recouvrement avant des erreurs appelée validation hybride [Mili, 85]. De plus, il semble qu'elle est plus efficace que la méthode des assertions exécutables [Jaoua and Mili, 90].

Mots clés: Recouvrement avant des erreurs, Tolérance aux fautes, Fiabilité, Spécification relationnelle, Relations difonctionnelles.

---

(1) This research is supported by grants from the Natural Science and Engineering Research Council of Canada (OGP0089762), Research Formation and help F.C.A.R. (1-3-04-3615-xxxx-2552), and from Laval University (A 268).

(2) Department of Computer Science, Laval University (Quebec), G1K 7P4.

(3) Department of Mathematics and Computer Science, Sherbrooke University (Quebec), J1K 2R1, Canada.

1. INTRODUCTION

The best way to build fault tolerant programs is to avoid introducing ad hoc procedures to find the origine of, detect or even recover, errors. In our present work, we present a constructive method for fault tolerant programming based on the principle that it is better to prevent than to cure. In hardware as in software, fault tolerance methodologies are based on error detection [Anderson and Lee, 1981], [Laprie, 1985], with backward error recovery [Aviziennis, 1985] or forward error recovery [Mili, 1985], [Jaoua and Mili, 1990].

In this paper we propose an approach for fault tolerance in which we do not try to detect errors but only use a general recovery routine as developed in [Jaoua, 1987]. This method may be applied to any fault tolerant program for which the time of error detection is longer than or equal to the time of error recovery. What is more interesting, in the special case of forward error recovery, because we use the information contained in the current program state, recovery routines may accelerate correct program termination. The proposed approach is a slight modification of the hybrid methodology proposed in [Mili, 85], and developed in [Jaoua, 87]. In the next section 2, we present the mathematical background of our work; in section 3, we describe the principles of the preventive method; in section 4, we introduce a p-reliability metric and use it to compare the reliability level of three particular merging programs: the first one is built without recovery, the second is built with error detection and recovery routines and the third program only uses a recovery routine. We conclude, in section 5, by some perspectives of application of this method in software engineering.

2. FUNDAMENTALS OF PROGRAM FAULT TOLERANCE

In our present work, we use the theoretical base of fault tolerance established in [Mili, 85]. The definitions we present are also derived from general references on fault tolerance [Anderson et al., 85] and [Laprie, 85]. We begin by giving some mathematical backgrounds.

2.1 Mathematical background

We place this study in the framework of a Tarski-like relational algebra [Tarski, 41].

2.1.1 Binary relations

Within a space S, which represents the structure of the information manipulated by a program, we define a binary relation as a subset of the Cartesian product S x S. An element of relation R is denoted (s, s'), where s is an argument (or input) of R, and s' is an image (or outpout) of s by R. In general, relation R is defined as follows:

$$R = \{(s,s') : i(s) \wedge b(s,s') \wedge o(s')\}$$

where i(s) is a unary predicate on the inputs of R, b(s,s'), is a binary predicate relating the outputs and inputs, and o(s') is a predicate on the outputs of R. A relational is a relation on the set of binary relations on S.

Remark: In this paper, binary relations are used as a program specification . [ ].

With a relation R are associated the following subsets of S:
the image set of $s \in S$, defined by $s.R = \{s' : (s,s') \in R\}$;
the domain, defined by $dom(R) = \{s : \exists s' : (s,s') \in R\}$;
the codomain (or range), defined by $cod(R) = \{s' : \exists s : (s,s') \in R\}$.

Let A be a subset of S, I(A) is the following subset of the identity relation I :

$$I(A) = \{(s,s') : s = s' \wedge s \in A\}.$$

2.1.2 Relationals

The intersection of relations R and R' is the relation

$$R \cap R' = \{(s,s') : (s,s') \in R \wedge (s,s') \in R'\}.$$

The union of relations R and R' is the relation

$$R \cup R' = \{(s,s') : (s,s') \in R \vee (s,s') \in R'\}.$$

The relative product (or composition) of relations R and R' is the relation

$$R \circ R' = \{(s,s') : \exists t \in S: (s,t) \in R \wedge (t,s') \in R'\}.$$

The inverse of relation R is the relation

$$R^{-1} = \{(s,s') : (s',s) \in R\}.$$

The kernel of relation R is the relation

$$K(R) = \{(s,s') : \varnothing \neq s'.R \subseteq s.R\}.$$

2.1.3 Difunctional relations

A relation $R \subseteq S \times S$ is difunctional if and only if it satisfies the condition $R \circ R^{-1} \circ R = R$. In [Jaoua et al., 90], we discovered that difunctionals are very general relations, and that in practive it is very rare to find non-difunctional specifications.

Remarks:
In [Jaoua et al., 90], difunctional relations are called "pseudo-invertible" or regular relations, later we have found the current name in [Riguet, 48].
In [Jaoua et al., 90], we prove that if R is difunctional then $K(R) = R \circ R^{-1}$.

2.2 Fundamentals of fault tolerance

Let p be a program on a space S, and let L be a label in the text of p. We denote the function of program p as [p]. Given that the label L may appear in the body of a loop, defining only the "state of the program at label L" may be ambiguous, because the label may be reached more than once. Instead, we define the state of the program at milestone m = (v,L) to designate the v^{th} visit to label L.

In the following, we present some useful definitions:

Past and future

Let p be a program on space S, and L be a label in the text of p. Given the milestone m = (v,L), we define two particular relations:

$PAST_{(v,L)}$ = {(s,s') : if the execution of p starts at state s, then it reaches milestone (v,L) at state s'}.

$FUTURE_{(v,L)}$ = {(s,s') : if the execution of p starts at milestone (v,L) on state s, then it terminates correctly at state s'}.

Failure, Error, Fault

Let P denotes the expected function of program p, and let s0 be an initial state for p. Let R be a specification defined on space S. Program p is supposed to be correct relativity to R, and $s0 \in dom(R)$.

Failure: Let us observe the execution of program p. If the program terminates at state P(s0), we say that p is executed with strict success. If the final state is in s0.R, we say that the program is executed with broad success. In the other cases, we say that we have observed a failure.

Error: Let m be a milestone of program p, let $PAST_m$ be the past relation at m, and let s_m be the program state at m. We say that there is an error at m for initial state s0, if s_m is different from $PAST_m(s0)$.

Fault: To define a fault, we should observe the path that program p has followed until milestone m. There is a fault in this path if the actual past function of p ($PAST_m$) at milestone m is different from its expected function.

3. PRINCIPLES OF THE PREVENTIVE METHODOLOGY

Preventive methodology is based on the idea that it is better to practice offensive rather than defensive programming. We apply this idea during the program design as well as program test phases. First, we built the program so that by its own structure it avoids failure. Second, to test its reliability, we insert several kinds of errors and verify, at execution, if the program terminates correctly within an acceptable time.

In the following sub-sections we first recall theoretical results, hybrid validation [Mili, 85] and executable assertion [Jaoua and Mili, 90] methods; second we give some characteristics of the preventive method.

3.1 Theoretical results of forward error recovery

In [Jaoua, 87], we find sufficient conditions for coherence and recoverability relatively to a difunctional specification R (defined in section 2), in the case of an initialised iterating program.

Let p = (i; L: w) be an initialised iterating program, and R the difunctional specification of program p. Then state s is correct relatively to specification R, vis-à-vis initial state s0, at milestone (v,L) if:

$$s.R = s0.R \text{ and } s \in dom([w] \cap R).$$

([w] denotes the function of statement w.)

Remarks:

(1) The condition s.R = s0.R says that we may replace s0 by s without modifying the final result of the program.

(2) The condition $s \in dom([w] \cap R)$ implies that, if the execution of w continues from state s, w may terminate in a correct state relatively to R. [ ]

The current state s at milestone (v,L) is recoverable relatively to R and vis-à-vis the initial state s0, if:

$$s.R = s0.R. \quad (1)$$

We also find that every routine that is correct relatively to the specification:

$$J = R \circ R^{-1} \circ I(dom([w] \cap R)) \quad (2)$$

is a correct recovery routine.

The condition s.R = s0.R is called critical information because it is a sufficient condition for s to be recoverable; the condition $s \in dom([w] \cap R)$ is called non-critical information, because even if we loose it, we may recover s by any routine r which is correct relatively to J.

3.2 Hybrid validation

The hybrid validation method consists of taking the following actions [Mili, 85]:

a) Proving that the critical information is preserved throughout the execution of the program (by Hoare logic, for exemple).

b) Protecting critical information by one of the following solutions:
   - saving the variables that are involved in the critical information in high-security memory [Schlichting and Schneider, 1980],
   - protecting the state against suspicious-looking modifications, by means of watchdog processor techniques [Mahmood et al, 1983].

c) Ensuring that the non-critical information information is recovered whenever it is damaged, using forward error recovery techniques.

Given the boolean function (detect) which returns true if and only if the current state is not in dom([w] ∩ R), and given the recovery routine (recover) which is correct with respect to the specification J, we augment the iterative program (i; while t do b) as follows:

```
p = begin
    i; (* initialisation *)
        while t do
            begin
            b; (* loop body *)
            if detect then recover (* added feature *)
            end
    end;
```

Such a program is said to be H-valid (H stands for : hybrid).

3.3 Executable assertions

In [Jaoua and Mili, 1990], we use executable assertions for error detection and damage assessment, so that, for an iterative program (i; while t do b), we obtain the following reliable version:

```
    i;
    ŝ := s; (* save the current state *)
L : while to do
    begin
    b;
    if not a((ŝ,s)) then recover;      (* if an error is detected then recover *)
    ŝ := s; (* save the current state *)
    end;
```

where (recover) is a recovery routine, and $a(\hat{s},s)$ is an executable assertion correct relatively to a specification

$$A = FUTURE \circ R^{-1} \circ R \circ FUTURE^{-1},$$

where R is a difunctional specification of the program p, and FUTURE is the future of the while-statement at milestone (v,L).

Remark: FUTURE = [while t do b], i.e. the functional abstraction of the iterative statement (while t do b).

3.4 The preventive methodology

In the two methods just described, we always check if the current state is erroneous (with the function (detect) or with the conditional statement (if not(a($\hat{s}$,s)) then recover). In both cases, the recovery routine and the function (detect) are designed to be independant from each other. We may think that it is interesting to find the non verified predicates while we are detecting errors, since such information may be used by the recovery routine to minimise the recovery time. This idea is rejected because it leads us to waste time in searching the cause of the errors. It also goes against the proposed forward recovery methods. Now, if we compare the specifications of the detect function and the recovery routine, we find a great similarity: in the hybrid validation method, because we assume that it is very easy to verify that critical information is preserved by the recovery routine, the essentiel task of this routine may be specified by the following specification J':

$$J' = \{(s,s') : s' \in \mathrm{dom}([w] \cap R)\},$$

while the function detect is specified by:

$$[\mathrm{detect}] = I(s \notin \mathrm{dom}([w] \cap R))$$

in the executable assertions method,

$$J' = \{(s,s') : a(\hat{s},s') = \mathrm{true}\},$$

while the function detect is specified by:

$$[\mathrm{detect}] = I(\mathrm{not}(a(\hat{s},s))).$$

In both cases the great symmetry of J' and [detect] leads us to believe that it is better to use for recovery the time we spend detecting errors. This principle is all the more interesting as the recovery routine may accelerate problem resolution whenever it is executed. Even if it seams very intrigue, in practive, for all tested applications we find that our fault tolerant programs make a speedy recovery without using detecting function. In the following we outline the steps of the preventive method.

Let p=(i;w) be an iterating program, where w = while t do b, and let R be the specification of p. The reliable version of p is the following program p':

```
p' = begin
        i;
        while t do
        begin
            b;
            recover
        end;
    end;
```

First we calculate the critical information, by solving the equation s. R = s0.R. Then, applying Hoare's logic, for example, we verify that the critical information is preserved. We may also implement some mechanism such as watchdog processor [Mahmood and al., 1983], or hight security memory [Schlichting and Schneider, 1980] to protect this information. Another solution consists to avoid manipulating the variables which are involved in the critical information, by using intermediate variables.

Second, (recover) must be chosen as the best routine which is correct relatively to the specification J.

Third, we use a "three steps testing" strategy:

a) Testing the program without (recover),
b) Testing the program with (recover) and without injecting errors,
c) Testing the program with (recover) and with injected errors on non-critical informations.

Remark: Generally, in step b), we verify whether the program is correct relatively to its specification, during execution. We also verify, whether (recover) routine is not erroneous. The original part of the program (i.e. without (recover)) and (recover) appear to mutually control each other. [ ]

4. RELIABILITY EVALUATION AND EXPERIMENTATION

The application of hardware reliability theories to software is often regarded skeptically (for example, see [Zelkowitz, 1978]. In [Conte and al., 1986], the software reliability is defined as: "the probability that there is no failure over n runs:

$$R(n) = 1 - (d_n/n)$$

where (d_n) is the number of defects over n runs".

The weakness of this definition of reliability is that it only measures the external behavior of the program. For example, even if we calculate the probability R(n) at the end of the testing step, it is not possible to deduce that the software will have the same reliability after its release. A solution is to find some criteria to recognize the end of the testing step, so that we will know exactly the future behavior of the software, with all its possibilities and limitations (in time, and in the kind of data it may process). In this work we defend the following ideas:

a) Building fault tolerant software with the preventive method increases its reliability.
b) Because the proposed programs are self-controled relatively to their own specification, if we assume that the specification is correct, then during the testing step, we find more existing errors than in the case of non recoverable programs.
c) We must adopt an offensive testing strategy: i.e. we do not test normal programs, but we inject several kinds of errors in the programs, and we verify if these programs terminate correctly within an acceptable time.
d) The preventive method permits the program to have a controled behaviour at the testing phase end: i.e. we never find unexpected defects after the testing phase.

From the third idea, we propose the following new definition of u-reliability of a fault tolerant program which measures its resistance to injected errors:

Definition 1: We say that a program p is u-reliable, for a time interval of length t and the input i, if and only if, whenever we introduce errors in its non-critical informations with probability u, than it terminates correctly its execution in a time shorter than t. [ ]

Definition 2: We say that a program p is relatively u-reliable, for a time interval of length t(n), and for q differents inputs, if and only if, whenever we introduce errors in its non-critical information with probability u, it terminates correctly its execution in a time shorter than t(n), for each execution, where n is the size of input i. [ ]

Definition 3: We say that a program p is absolutely u-reliable, for a time interval of length t(n) and a specification R, if and only if, for each input i of dom(R), whenever we introduce errors in the non-critical information of p with probability u, it terminates correctly its execution in a time shorter than t(n), where n is the size of the input i. [ ]

Remark: These definitions give us an approximate idea of the reliability level of the program. However, we are beginning to study a more accurate mathematical calculus of reliability, based on probabilistic errors. [ ]

Using a particular problem, in the following subsection, we give a detailed description of all the program construction steps using the preventive method.

4.1 Problem definition

We propose to write a program (called Pseudo-merge) which realises a special merge of two sorted arrays T1 and T2 into a third array T3: we assume that the last element of the second array is always greater than the last element of the first array, and that we always ignore all the elements of T2 which are greater than the last element of T1.

Example: Let T1 = [2, 7, 9, 11, 20], and
T2 = [1, 3, 6, 8, 9, 15, 56, 67],
then T3 = [1, 2, 3, 6, 7, 8, 9, 9, 11, 15, 20]. [ ]

For all the following ten tests, we will take as inputs the same values of T1 and T2 as those of the example, when the result is correct we obtain the same value of T3 as the example else we obtain a different value.

In the following sub-sections we propose a formal description of space S and specification R of Pseudo_merge:

4.1.1 Space definition (S)

S is the Cartesian product of several sets. We represent the product

integer x integer x ... x integer (n times)

by "array [1..n] of integers";
we also represent (real x integer x char) by

Set
A: real;
B: integer;
C: char
end

If we use this formalism, we find

S = Set
T1: array [1..n] of integer; (first input array)
T2: array [1..m] of integer; (second input array)
T3: array [1..p] of integer; (output array)

sub
  sorted (T1[1..n]);
  sorted (T2[1..m]);
  T1[n] <= T2[m];
  T2[k+1] >= T1[n];
  k(s) < m(s)
End;

where sub ... end section contains the restrictions that set S must verify, n, m and p represent respectively the number of the elements of T1, T2 and T3, and k is the greatest index such that T2[k] < T1[n], and sorted (T[1..m]) the following predicate:

sorted (T[1..m]) is true if and only if T[1] <= T[2] <= ... <= T[m].

4.1.2 Relational specification (R)

4.1.2.1 Notations

Let "union", and "perm" be the following predicates:

(union (T1[a1..b1], T2[a2..b2]) = T3[a3..b3]) is true,

if and only if the set of the elements of T3 between the indices a3 and b3 is equal to the union of the elements of T1 located between a1 and b1 and the elements of T2 located between a2 and b2, and

perm (T1[a1..b1], T2[a2..b2] is true,

if and only if the set of the elements of T1 located between a1 and b1 is equal to the set of the elements of T2 located between a2 and b2. For a variable v of the space S, v(s) represents its input value, and v(s'), its output value.

4.1.2.2 Problem specification (R)

The specification of the problem R is equal to the conjunction of two elementary relations:
$R = R1 \cap R2$,

where

R1 = {(s,s') :

p(s') = k(s) + n(s)

&

perm (T3(s') [1..k(s) + n(s)], union (T1(s) [1..n(s)], T2(s) [1..k(s)])

}

is a difunctional relation because, when we calculate R1 o $R1^{-1}$ o R1, we find exactly R1, and

$$R2 = \{(s,s') : \text{sorted}(T3(s')\ [1..p(s')])\ \}$$

is a unary relation on the output states, so R2 is difunctional [Jaoua et al., 1990]. Because R is the conjunction of two difunctional relations, we deduce that R is also a difunctional [Jaoua et al., 1990].

4.2 Critical information calculus

Critical information is formulated by:

$$\underline{(s0,s) \in R \text{ o } R^{-1}}$$

where s0 is the initial state and s the current state.

$$R = R1 \cap R2$$

$$\Rightarrow\ R = \{(s,s') : \quad p(s') = k(s) + n(s)$$

$$\&$$

$$\text{perm } (T3(s')\ [1..k(s) + n(s)],\ \text{union } (T1(s)\ [1..n(s)],\ T2(s)\ [1..k(s)])$$

$$\&$$

$$\text{sorted } (T3(s')\ [1..p(s')])$$

$$\}.$$

We first calculate:

$$R \text{ o } R^{-1} = \{(s,s') : \exists\, s'' : (s,s'') \in R\ \&\ (s'',s') \in R^{-1}\},$$

$$= \{(s,s') : \exists\, s'' : (s,s'') \in R\ \&\ (s',s'') \in R\},$$

$$= \{(s,s') : \exists\, s'':$$

$$p(s'') = k(s) + n(s)$$

$$\&$$

$$\text{perm } (T3(s'')\ [1..k(s) + n(s)],\ \text{union } (T1(s)\ [1..n(s)],\ T2(s)\ [1..k(s)])$$

$$\&$$

$$\text{sorted } (T3(s'')\ [1..p(s'')])$$

$$\&$$

$$p(s'') = k(s') + n(s')$$

$$\&$$

$$\text{perm } (T3(s'')\ [1..k(s') + n(s')],\ \text{union } (T1(s')\ [1..n(s')],\ T2(s')\ [1..k(s')])$$

$$\&$$

$$\text{sorted } (T3(s'')\ [1..p(s'')])$$

$$\}$$

=>

$$R \circ R^{-1} = \{(s,s') : k(s') + n(s') = k(s) + n(s) \quad (1)$$

$$\&$$

$$\text{union } (T1(s)\ [1..n(s)],\ T2(s)\ [1..k(s)]) =$$

$$\text{union } (T1(s')\ [1..n(s')],\ T2(s')\ [1..k(s')]) \quad (2)$$

$$\};$$

=> the critical information is:

$$k(s0) + n(s0) = k(s) + n(s)$$

$$\&$$

$$\text{union } (T1(s0)\ [1..n(s0)],\ T2(s0)\ [1..k(s0)]) =$$

$$\text{union } (T1(s)\ [1..n(s)],\ T2(s)\ [1..k(s)])$$

4.3 The non fault tolerant program Pseudo_merge

Here, we only show the program procedure Pseudo_merge, we assume that initially, T1(s0) and T2(s0) are in S:

```
pseudo_merge()
{
    k1=k2=k3=0;  /* k1, k2 and k3 respectively index T1, T2 and T3 */
    while (k1<N)
    {/* We store the smallest of T1[k1] and T2[k2] in T3[k3] */
        if (T1[k1] <= T2[k2]
        {
            T3[k3] = T1[k1];
            k1 += 1;
        }
        else
        {
            T3[k3] = T2[k2];
            k2 += 1;
        }
        k3 += 1;
    }
}
```

The critical section that manipulates the critical information is empty, because we never modify T1 or T2, but we only compare or store their values. Hence the program preserve critical information, by construction. For more safety, we may use two copies of T1 and T2 as intermediate arrays.

4.4 Detect function construction

The function detect is specified by:

$$s \in dom([w] \cap R)$$

where [w] is the functional abstraction of the iterative statement.

$$
\begin{aligned}
[w] = \{(s,s') :\ & perm\ (T3\ [k3(s)..k3(s')],\\
& union\ (T1(s)\ [k1(s)..k1(s')],\ T2(s)\ [k2(s)..k2(s')])\\
& \&\\
& sorted\ (T3(s')\ [k3(s)..k3(s')]\\
& \&\\
& T3(s')\ [1..k3(s) - 1] = T3(s)\ [1..k3(s) - 1]\\
& \&\\
& k3(s') = k3(s) + k1(s') + k2(s') - k1(s) - k2(s)\\
& \&\\
& T2(s)\ [k2(s') + 1] >= T1(s)\ [k1(s')]\\
& \&\\
& k1(s') = n\\
& \&\\
& k2(s') < m\\
& \&\\
& 0 <= k1(s) <= n\\
& \&\\
& 0 <= k2(s) <= m-1\\
& \&\\
& 0 <= k3(s) <= n+m-1\\
\};&
\end{aligned}
$$

=>

$$
\begin{aligned}
[w] \cap R = \{(s,s') :\ & \\
& p(s') = k(s) + n(s)\\
& \&\\
& perm\ (T3(s')\ [1..k(s) + n(s)],\ union\ (T1(s)\ [1..n(s)],\ T2(s)\ [1..k(s)])
\end{aligned}
$$

&
sorted (T3(s') [1..p(s')])
&
T3(s') [1..k3(s) - 1] = T3(s) [1..k3(s) - 1]
&
perm (T3(s') [k3(s)..k3(s')],
union (T1(s) [k1(s)..k1(s')], T2(s) [k2(s)..k2(s')])
&
sorted (T3(s') [k3(s)..k3(s')]
&
k3(s') = k3(s) + k1(s') + k2(s') - k1(s) - k2(s)
&
T2(s) [k2(s') + 1] >= T1(s) [k1(s')]
&
k1(s') = n
&
k2(s') = k(s)
&
0 <= k1(s) <= n
&
0 <= k2(s) <= m-1
&
0 <= k3(s) <= n+m-1

};

=> D = dom([w] ∩ R) = {s :
perm (T3(s) [1..k3(s) - 1],
union (T1(s) [1..k1(s) - 1], T2(s) [1..k2(s) = 1] (1)
&
sorted (T3(s) [1..k3(s) - 1]) (2)
&
k3(s) = k1(s) + k2(s) (3)
&
0 <= k1(s) <= n (4)
&
0 <= k2(s) <= m-1 (5)
};

=> the specification of the function detect is:

$$DT = \{(s,s') : detect(s') = (s \notin D\}.$$

=> We propose the following function detect:

```
char detect ( )
{
    int a1,a2,a3;
    char error;
    a1=a2=a3=0; error=0;
    while (! error && a1<=k1 && a2<=k2 && a3<=k3 && (a1<=N) && (a2<M))
    {
      if  (T3[a3] <= T3[a3+1])        /*  we verify condition (2) else
                                          we detect an error */
      {   /* we verify condition (1) else we detect an error */
          if          (T3[a3]==T1[a1])      {a1 += 1; a3 += 1;}
          else if     (T3[a3]==T2[a2])      {a2 += 1; a3 += 1;}
          else error = 1;
      }
      else error = 1;
    }
    / * we verify the conditions (3), (4) and (5) * /
    return (error | | (k1!=a1) | | (k2!=a2) | | (k3!=a3));
}
```

4.5 The recovery routine construction

We first begin by calculating the specification of the recovery routine J:

$$J = R \circ R^{-1} \circ I(dom([w] \cap R))$$

and we find:

$$J \{(s,s') : perm(T3(s')[1..k3(s')-1], union(T1(s)[1..k1(s')-1], T2(s)[1..k2(s')-1])) \quad (1')$$

$$\& \; k3(s') = k1(s') + k2(s') \quad (2')$$

$$\& \; sorted(t3(s')\,[1..k3(s')-1] \quad (3')$$

$$\& \; 0 <= k1(s') <= n \quad (4')$$

$$\& \; 0 <= k2(s') <= m-1 \quad (5')$$

& /* critical conditions */

};

We assume that critical conditions (i.e. the preservation of critical information) are always verified by the program. Hence every recovery routine which is correct relatively to J' (J without critical conditions) is a correct recovery routine. Here we present one such recovery routine (**recover**).

```
recover ( )
{  /* Recover studies the current state of the program, and changes it into a correct state; i.e. a
state that verifies conditions (1'), (2'), (3'), (4') and (5')  */
    int a1,a2,a3;
    char correct_state;
    a1=a2=a3=0; correct_state=0;
    while (! correct_state && (a1<N) && (a2<M))
    {
        if (T3[a3] <= T3[a3+1])
        {
            if          (T3[a3]==T1[a1]) {a1 += 1; a3 += 1;}
            else if     (T3[a3]==T2[a2]) {a2 += 1; a3 += 1;}
            else correct_state = 1;
        }
        else correct_state = 1;
    }
    k1=a1; k1=a2; k3=a3;
}
```

Remark: The logical structures of the proposed recovery routine and the function detect are almost the same. We say even notice that the structure of the recovery routine is simpler than the structure of error detection. While **detect** tries to check if the current state is erroneous, **recover** tries to analyse the current state to map it into a correct state. [ ]

4.6 Testing strategy

In the following, we give experimental results with a sample of injected errors in three versions of "pseudo_merge" program: without error recovery and detection, with error detection and recovery, and finally with recovery only. The programs were executed on IBM PS2, 16 Mhz with Quick C 2.0 compiler, Microsoft.

Test 1: We begin by testing the non fault tolerant original C program which must execute correctly without error injection. Here, we do not use a recovery routine, or any function to detect errors.

Results of test 1:

The result is correct.
Run time: 110 milliseconds.

Test 2: We inject an error on the variable k1 with a probability 0.8, in the non fault tolerant initial program.

```
pseudo_merge ( )
{
    k1=k2=k3=0;
    while (k1<N)
    {
        if (T1[k1] <= T2[k2])
        {
            T3[k3] = T1[k1_;
        k1 = k1 + (((rand ( ) % 100) < 99 ? (rand ( ) % 5) : 1);
            /* error injection on the value of k1 with a probability 0.99 */
            /* we add an erroneous value to k1 with a probability 0.90 */
        }
        else
        {
            T3[k3] = T2[k2];
            k2 += 1;
        }
        k3 += 1;
    }
}
```

Remark: rand is the standard random function of the C language, it gives a random value between 0 and 32767. [ ]

Results of test 2:

The result is not correct.
Run time: 110 milliseconds. [ ]

Test 3: We inject errors on k1 with a probability 0.99, but we detect and recover existing erroneous states of the program.

```
pseudo_merge ( )
{
    k1=k2=k3=0;
    while (k1<N)
    {
        if (T1[k1] <= T2[k2])
        {
            T3[k3] = T1[k1];
        k1 = k1 + (((rand ( ) % 100 < 99) ? (rand ( ) % 5) : 1);
        }
        else
        {
            T3[k3] = T2[k2];
            k2 += 1;
        }
        k3 += 1;
    if (detect ( )) recover ( );
        /* detection and error recovery */
    }
}
```

Results of test 3:

The result is correct.
Run time: 170 milliseconds. [ ]

Test 4: Same error injection as in test3, but this time we systematically recover the current state of the program without trying to detect errors.

```
pseudo_merge ( )
{
    k1=k2=k3=0;
    while (k1<N)
    {
        if (T1[k1] <= T2[k2])
        {
            T3[k3] = T1[k1];
        k1 = k1 + (((rand ( ) % 100) < 99) ? (rand ( ) % 5) : 1);
        }
        else
        {
            T3[k3] = T2[k2];
            k2 += 1;
        }
        k3 += 1;
    recover ( );
        /* We always recover the current state of the program */
    }
}
```

Results of test 4:

The result is correct.
Run time: 110 milliseconds.

Remark: Without trying to detect errors, our program is faster (see the result of test 3), it runs in exactly the same time as the non-fault-tolerant version of Pseudo_merge. [ ]

Test 5: Error injections on the three indices k1, k2 and k3, with probability 99%, in the original program, without error detection or recovery.

```
pseudo_merge ( )
{
    k1=k2=k3=0;
    while (k1<N)
    {
        if (T1[k1] <= T2[k2])
        {
            T3[k3] = T1[k1];
        k1 = k1 + (((rand ( ) % 100 < 99) ? (rand ( ) % 5) : 1);
        }
        else
        {
            T3[k3] = T2[k2];
        k2 = k2 + (((rand ( ) % 100 < 99) ? (rand ( ) % 5) : 1);
        }
    k3 = k3 + (((rand ( ) % 100 < 99) ? (rand ( ) % 5) : 1);
    }
}
```

Results of test 5:

The result is not correct.
Run time : 110 milliseconds. [ ]

Test 6: The same errors as in test 5 are injected, but we only add the statement if detect then recover, as we can observe in the following program:

```
pseudo_merge ( )
{
    k1=k2=k3=0;
    while (k1<N)
    {
        if (T1[k1] <= T2[k2]
        {
            T3[k3] = T1[k1];
        k1 = k1 + (((rand ( ) % 100) < 99) ? (rand ( ) % 5) : 1);
            /* Error injection on the value of k1, with 0.99 probability */
        }
        else
        {
            T3[k3] = T2[k2];
        k2 = k2 + (((rand ( ) % 100) < 99) ? (rand ( ) % 5) : 1);
            /* Error injection on the value of k2, with 0.99 probability */
        }
    k3 = k3 + (((rand ( ) % 100) < 99) ? (rand ( ) % 5) : 1);
            /* Error injection on the value of k2, with 0.99 probability */
    if (detect ( )) recover ( );
            /* Error detection and recovery */
    }
}
```

Results of test 6:

The result is correct.
Run-time: 170 milliseconds. [ ]

Test 7: The same errors as in test 5 are injected, but we only add the statement: recover, as we can observe in the following program:

```
pseudo_merge ( )
{
    k1=k2=k3=0;
    while (k1<N)
    {
        if (T1[k1] <= T2[k2])
        {
            T3[k3] = T1[k1];
        k1 = k1 + (((rand ( ) % 100) < 99) ? (rand ( ) % 5) : 1);
        }
        else
        {
            T3[k3] = T2[k2];
        k2 = k2 + (((rand ( ) % 100) < 99) ? (rand ( ) % 5) : 1);
        }
    k3 = k3 + (((rand ( ) % 100) < 99) ? (rand ( ) % 5) : 1);
    recover ( );
    }
}
```

Results of test 7:

The result is correct.
Run time: 110 milliseconds.

Remark: Without trying to detect errors, this program is faster (see the result of test 6), it is also the same run time as the non-fault-tolerant version of Pseudo_merge. [ ]

Test 8: Error injections on the three indices k1, k2, k3 with 0.99 % and on the output array T3 with 90%, in the original program, without error detection or recovery.

```
pseudo_merge ( )
{
    k1=k2=k3=0;
    while (k1<N)
    {
        if (T1[k1] <= T2[k2])
        {
        T3[k3] = ((rand ( ) % 100) < 90) ? rand ( ) : T1[k1];
            /* Error injection on T3 with 0.90 probability */
        k1 = k1 + (((rand ( ) % 100) < 99 ? (rand ( ) % 5) : 1);
            /* Error injection on k1 */
        }
        else
        {
        T3[k3] = ((rand ( ) % 100) < 90) ? rand ( ) : T2[k2];
            /* Error injection on T3 with 0.90 probability */
        k2 = k2 + (((rand ( ) % 100) < 99) ? (rand ( ) % 5) : 1);
            /* Error injection on k2 with 0.99 probability */
        }
    k3 = k3 + (((rand ( ) % 100) < 99) ? (rand ( ) % 5) : 1);
            /* Error injection on k3 with 0.99 probability */
    }
}
```

Results of test 8:

The result is not correct.
Run time: 110 milliseconds. [ ]

Test 9: Same injected errors as in test 8, but we add the following statement: if detect then recover.

```
pseudo_merge ( )
{
    k1=k2=k3=0;
    while (k1<N)
    {
        if (T1[k1] <= T2[k2])
        {
        T3[k3] = ((rand ( ) % 100) < 90) ? rand ( ) : T1[k1];
        k1 = k1 + (((rand ( ) % 100 < 99) ? (rand ( ) % 5) :1;
        }
        else
        {
        T3[k3] = ((rand ( ) % 100) < 90) ? rand ( ) : T2[k2];
        k2 = k2 + (((rand ( ) % 100) < 99) ? (rand ( ) % 5) : 1);
        }
    k3 = k3 + (((rand ( ) % 100) < 99) ? (rand ( ) % 5) : 1);
    if (detect ( )) recover ( );
        /* If error detected then we recover the state of the program */
    }
}
```

Results of test 9:

The result is correct.
Time: 280 milliseconds. [ ]

Test 10: Same errors injected as in test 9 (and test 9), but we recover the current state of the program without detecting errors.

```
pseudo_merge ( )
{
    k1=k2=k3=0;
    while (k1<N)
    {
        if (T1[k1] <= T2[k2])
        {
        T3[k3] = ((rand ( ) % 100) < 90) ? rand ( ) : T1[k1];
        k1 = k1 + (((rand ( ) % 100) < 99) ? (rand ( ) % 5) : 1);
        }
        else
        {
        T3[k3] = ((rand ( ) % 100) < 90) ? rand ( ) : T2[k2];
            k2 = k2 + (((rand ( ) % 100) < 99) ? (rand ( ) % 5) : 1);
        }
    k3 = k3 + (((rand ( ) % 100) < 99) ? (rand ( ) % 5) : 1);
    recover ( );
        /* Error recovery */
    }
}
```

Results of test 10:

The result is correct.
Time: 160 milliseconds. [ ]

Remark: Without error detection, this program is (7/4) times as fast as the program of test 9. For t = 160 milliseconds the fault tolerant verison is 90%-reliable relatively to the current inputs.

5. CONCLUSION

In this paper we propose a preventive methodology to built fault tolerant programs. This method may be viewed as an improvement of the hybrid validation methodology [Mili, 1985]. However, it seams that the preventive method is more efficient when the recovery routine participates in the problem resolution and when it is possible to measure the "distance" between the current state and the final expected state of the program.

The ten experimental results which are presented in this paper are good indicators of the preventive methodology strength. However, to find more affirmative conclusions, we are planifying several experimentations, on different kinds of applications (small, medium and big). We are also orienting our effort on mathematical proofs of absolutely u-reliable programs, and in a case by case study of fault-tolerant programs complexity. We try to give exact answers to questions as: "how much time does the program run during recovery ?", "is there some heuristics to find a better recovery routine ?", "is it possible to find a natural program structure which is both fault tolerant and as efficient as a non-fault tolerant version of the same program ?". Our purpose is to discover the pertinent program parameters with which we may increase its fault resistance without adding excessive human overhead during its development.

Acknowledgment

The authors are grateful to Dr. M. Beaudry (University of Sherbrooke) and A. Mili (University of Tunis) for their comments and suggestions for improvement.

REFERENCES

[Anderson, 1981] Anderson ,T. & Lee, A. Fault Tolerance, Principles and Practice. Prentice Hall International Series - 1981.

[Anderson, 1981] Anderson, T. Can Design Faults be Tolerated ? Colloquium sponsored by: Le Centre de Recherche Informatique de Montréal (CRIM), Montreal Crim, 14 juin 1985.

[Avizienies, 1985] Avizienies, A. Fault Tolerance: an Overview. Colloquium sponsored by: Le Centre de Recherche Informatique de Montréal (CRIM), 14 juin 1985.

[Conte et al.] Conte, S.D., Dunsmore, H.E. and Shen, V.Y. Software Engineering Metrics and Models. The Benjamin/Cummings Publishing Company Inc., 1986.

[Dijkstra, 1976] Dijkstra, E. A Discipline of Programming. Englewood Cliffs (NJ) : Prentice-Hall, 1976.

[Gries, 1981] Gries, D. The Science of Programming. Springer Verlag, 1981.

[Jaoua, 1987] Jaoua, A. Recouvrement Avant de Programmes sous les Hypothèses de Spécification Déterministe et non Déterministe. Dissertation of Doctorat ès Sciences, Université Paul Sabatier, Toulouse, France. Defended December 1987.

[Jaoua, 1990] Jaoua, A. & Mili, A. The Use of Executable Assertion for Error Detection and Damage Assessement. Journal of Systems and Software, April 1990.

[Mahmood et al.] Mahmoud, A., McCluskey, E.J. and Lu, David J. Concurrent Fault Detection using a Watchdog Processor and Assertions, June 1983. Private Correspondance. Stanford University.

[Mili, 1985] Mili, A. Toward a Theory of Forward Error Recovery. IEEE Trans. on Soft. Eng., August 1985.

[Mili et al., 1987] Mili, A., Guemara, S., Jaoua, A. and Torres, P. On the Use of Executable Assertions in Structured Programs. The Journal of Systems and Software, March 1987.

[Riguet, 1948] Riguet, J. Relations Binaires, Fermetures et Correspondances de Galois. Bulletins de la Société Mathématique de France, 1948, pp. 114-155.

[Schichting and Schneider, 1980] Schichting and Schneider. Verification of Fault Tolerance Software. Cornell University, Computer Science Department, Report TR-80-446, November 1980.

[Tarski, 1941] Tarski, A. On the Calculus of Relations. The Journal of Symbolic Logic 6(3), September 1941, pp. 73-89.

[Zelkowitz, 1978] Zelkowitz, M.V. Perspectives on Software Engineering. ACM Computing Surveys 10 (2), 1978, pp. 197-216.