# Application Semantic Driven Assertions toward Fault Tolerant Computing

**Goutam Kumar Saha**

Member ACM

Scientist-F, Centre for Development of Advanced Computing, Kolkata, India.
Mail to: CA-2 / 4B, CPM Party Office Road, Baguiati, DB Nagar,
Kolkata 700059, West Bengal, India. gsaha@acm.org <sahagk@gmail.com>

***Based on semantics of an application processing logic, we find out the most critical and sensitive parts of an application and we derive set of conditions or assertions among the various diagnostic checkpoint variables and we enhance the processing logic to enable it to detect run-time various operational or environmental faults toward fault tolerant computing. This paper examines how a single-version algorithm can establish software based fault tolerance by designing in thoughtful software based execution-time checks in a computing application. The algorithm developed here relies on various assertions that are derived from the semantics of an application. Various diagnostic assertive checkpoints have been derived based on an application's semantics. This work is not intended to correct bit-errors using conventional error correction codes. Errors have been detected through checkpoints and periodical execution of an application with known test data and verification of observed result with known result thereof. Electrical transients or small particles hitting the circuit, often cause random errors or faults in data and program flow. The manuscript describes an algorithm that allows the detection and recovery of transient or operational failures in software on a specific problem, just by using one version of a software program running on just one machine. This approach does not aim to tolerate software design bugs. This algorithmic approach uses various run-time signatures and validation thereof in order to detect faults.***

## 1. Introduction

Because of tremendous increase in complexity of computer systems and enormous increase in the computing power of computers, fault – tolerant design of application system has

become essential for gaining high accuracy and reliability in computation. Short duration noises pose a formidable threat to a reliable processing unit. Noises change the stored information bits or affect the program counter, program flow. A failure is said to occur when a system does not meet its specifications. A *valid system is one, which the system operates in accordance with the specifications.* An *erroneous state* of a system is an internal state, which could lead to failure despite a sequence of valid transitions. An error triggers a transition from normal state to an erroneous. *An error in hardware or software component of the system is referred to as a fault in the system.* A *transient fault is not reproducible under controlled conditions*. A *permanent fault* is one that requires corrective action for its removal, and, restoration of normal system operation. A permanent *fault* remains unless it is removed by some external agency. Whereas a *transient fault* eventually disappears without any apparent intervention. Though it may seem that permanent faults are more severe, from an engineering perspective, they are much easier to diagnose and handle. Most problematic type of transient fault is the intermittent fault that occurs often unpredictably. *Errors in program flow can be caused by changes in either the internal registers of the processor unit or a memory bit that is part of the program instructions. Fault Tolerance* is the ability of a system to perform its specified function correctly even if the internal faults exist. The purpose of fault tolerance is to increase the dependability of a system. In other words, a fault is the root cause of a failure. An error is merely the symptom of a fault. A fault may not necessarily result in an error, but the same fault may result in multiple errors. Similarly, a single error may lead to multiple errors. Depending on duration, faults can be classified as permanent or transient. In other words, fault tolerance *is the study of errors, faults and failures and of methods for enabling systems to continue normal operation.* Software fault tolerance *is to detect an erroneous task as soon as possible and to halt the task to prevent error propagation for protecting the system complete disruption. After the necessary recoveries, again it is to continue normal operations.* ***Assertion is a statement of conditions or relationships between variables that should exist instantaneously at a particular point in a program.***

The proposed approach does not consider the issue of eliminating software bugs. It is assumed that the code is correct and the erroneous behavior is only due to transient faults

affecting the computing system. The proposed approach is aimed to detect soft errors during the run time of a specific application for solving a series of quadratic equations towards fail-stop kind of fault tolerance. The objective of this paper is to describe an assertion - based annotation of code as an approach to provide detection of and recovery from transient errors. The programmer has to manually insert correctness conditions for the data in the program and inserting a kind of an encoding of the program trace. Programmer has to add check- variables (one for each operation) to the program, and has to update these variables after each operation to reflect a post condition of the operation. Additional check-variables are to count how many times each block has been executed in order to detect the correctness of the program flow or program control. Programmer manually generates a checker, which controls whether checkpoint variables satisfy certain bounds, and whether the block-execution counters satisfy certain bounds as well. If errors are detected, the checker calls an appropriate recovery procedure. In brief, this paper shows how a programmer can provide some tolerance to hardware transients by inserting counters in an application's iteration loops and then, after each iteration, checking to determine if the counters have been incremented in a consistent way. This approach can verify whether hardware transients have corrupted a program itself by making a duplicate copy of it and then comparing the two copies or, alternately, by using checksum on a single copy. The type of errors changing the execution flow due to the transients affected statements of type e.g., loops, procedure calls, tests, returns etc., can be detected. Again, the errors changing the operation to be performed by the statement, without the code execution flow, caused by erroneous statements affecting data only (e.g., arithmetic expression computations, assignments etc.) Are detected by periodical execution of the application with known input data and then by comparing the result with the pre-computed known result.

## 2. Background

External interference like Electrical Fast Transients often cause alteration of the stored bit patterns or enable false *interrupt* resulting in faulty *program flow, false branching.* The conventional error codes like Parity bit, CRC, Hamming Codes are not free from limitations. Though they are capable of detecting multiple errors but most of them cannot

correct multiple errors. Again, software implementation of such codes suffers from an unaffordable redundancy with both memory space and execution time. The common arrangements for achieving fault tolerance consist of providing some form of protective redundancy in the system. Redundancy can be applied either in hardware (for example by using a duplex or a higher module replication), or in the software (as the N version programming redundancy of [1]). As in a redundant system, replication of at least twice the hardware or software components is required, and then the cost of manufacturing fault tolerant systems can be expected to increase substantially [1]. The N version programming is basically based on design diversification running concurrently on many machines and aims to tolerate software bugs. Here, we need to vote the N number of results in order to find the result with majority and it is considered as the correct one. If no majority in result is found then the N versions technique fails. Here, the software redundancy is N and the hardware redundancy is also N. The conventional idea of N version programming [1,2] is to detect the software design bugs. However, the reality of development is such that developing one version with reasonable quality and budget is itself a challenge. Developing three or more is out of the question for most scientific and commercial development. Again, the *Duplex Time Redundancy Scheme* (DTRS) [3], can not mask single transient faults and a job that has a discrepancy in the results produced by the different versions, must be retried for achieving fault tolerance. Simple periodic tasks have been discussed in [4]. The space redundancy should not be very high as in [1,2,3,4] because in many of today's real-time systems, memory constraints are still a bottleneck [5,6] and therefore need to be accounted for. A set of carefully chosen software error detection techniques including Assertions [7], Algorithm Based Fault Tolerance (ABFT) [8], Control Flow Checking [9], procedure duplication [10] are suitable to achieve a high degree of safe behavior in ordinary computers by complementing the intrinsic Error Detection Mechanisms (Edemas) of the system (exceptions, memory protection, etc). In [11] it has been shown that it is possible to achieve an adaptive infrastructure to support different levels of availability requirements in a network environment. In [12] it has been shown how to detect transient errors and how to recover on using triple copies of an enhanced application using *single version* only. The single-version software needs only one enhanced version of software

running on one machine. This proposed single-version software technique use the idea of including checkpoints in an algorithm in order to verify the intermediate calculations and, eventually, the correctness of the final results. The objective of this fault tolerant software is to detect an error as quickly as possible, before it has a chance to do any damage. This paper describes the approach to software fault tolerance by preventing a task failure to lead to the system complete disruption. The objective *of this approach* is to detect an erroneous task as soon as possible and is to halt the erroneous task for preventing error propagation. In this proposed technique the *run-time or observed values of different flags, variables are automatically verified and validated against their theoretical or expected values.* Any discrepancy between the expected and the run-time observed values of a variable is detected quickly and an error-routine is invoked for taking necessary recovery *action.* After recovery, the application is re-executed for gaining fault tolerance through fail-stop. All these activities are carried out automatically by the *built-in procedures,* without any need of human interventions. The proposed technique does not need any major replication (as in [1,2]) of the application program because it uses only an affordable number of extra *instructions* instead of using many versions as in [1]. Here, the space *redundancy is affordable*. Because this *proposed technique does not need any more extra versions except the enhanced version of the application program.* The *time redundancy or the run-time overhead is also affordable* because here, only the enhanced version of the application program needs to be executed for delivering the reliable result. In this paper, the proposed technique has been demonstrated thoroughly through a computational problem like, solving a series of quadratic equations. During the computation, it can verify if any number of errors or faults has occurred in the data or program flow and then it takes necessary recovery action. It does not use any conventional error codes. We have demonstrated the approach on a specific problem on solving a series of quadratic equations. The objective of the paper is to find out a low cost and an efficient solution towards faults tolerant computing. The executable code size increases by 1.6 times and execution slows down by 1.52 times with respect to the basic processing logic. For further readings, works in [13-22] must also be read.

### 3.The Application Semantic Fault Tolerance

The basic computing logic (as stated in *algorithm-1*) is enhanced with some more effective logic that rely on an application semantic for attaining higher fault tolerance in this computing problem which needs higher accuracy. The enhanced logic for a reliable computation for solving a *series of quadratic equations* is stated in *algorithm-2*.

***The Basic Algorithm:***

*Algorithm-1*: It states the *basic steps* required for solving a series of quadratic equations.

*Step 1*. Read: $T_Q$ /* Total number of quadratic equations */

*Step 2*. Repeat for $I = 1$ to $T_Q$ by 1:

Read: $C_A[I], C_B[I], C_C[I]$

/* Read Coefficients */

[End of Loop]

*Step 3*. Repeat for $I = 1$ to $T_Q$ by 1:

Set $R_D := C_B[I]^2 - 4\, C_A[I]\, C_C[I]$.

If $R_D > 0$, *then:*

Set $X_1 := ( - C_B[I] + \text{sqrt}(R_D)) / 2\, C_A[I]$

Set $X_2 := ( - C_B[I] - \text{sqrt}(R_D)) / 2\, C_A[I]$

Write: $X_1[I], X_2[I]$.

Else if $R_D = = 0$, *then:*

Set $X[I] := - C_B[I] / 2\, C_A[I]$.

Write: *'Unique Solution', X[I].*

Else:

Write: *'No Real Solutions.'*

[End of If structure]

[End of Step-3 Loop]

*Step 4. Exit*

{End of Basic Processing *Algorithm-1*}

## *The Enhanced Algorithm towards Fault Tolerance*:

***Enhanced Algorithm-2*.** It describes the enhanced logic using assertions based on application semantics, in order to achieve transient fault tolerance and thus ensures error free computing session. It calls a procedure called *EXAM-EXAMCODE* which in turn calls an error or fault checking routine namely, EXAMCODE.

*Step 1*. Read: $T_Q$ /* Reads the total number of quadratic equations to be solved*/

$CT_Q := T_Q$ /*Variable $CT_Q$ stores the value of $T_Q$ */

*Step2.* Set *GTB1:=0, GTB2:=0, GTB3:=0,*

*GTB4:=0, GTB5:=0, GTPF1:=0,GTPF2:=0, GTPF3:=0, GTPF4:=0,*

*GTPF5:=0, GTS:=0, GTBI:=0*

/*All these global variables are initialized to zero. These variables are used in various program blocks as various checkpoints. The variable *I* is also a global one. */

*Step 3*. Repeat for $I = 1$ to $T_Q$ by 1:

Read: $C_A[I], C_B[I], C_C[I]$

/* Read the coefficients of an equation*/

$GTB1 = GTB1 + 1$

/* Increment the variables *GTB1, GTS* after reading one set of coefficients, at the program block say, *B1* */

$GTS = GTS + 1$

$GTPF1 = 1$

$GSABC[I] = C_A[I] + C_B[I] + C_C[I]$

/**GSABC* variable stores the sum of all the three coefficients of an equation*/

If $I > CT_Q$, then:

/*Prevents infinite loop at step-3 */

Goto *Err_B1*.

/* Check validity of the run-time values of both the variables namely, $CT_Q$ and $I$ */

[End of If structure]

[ End of *Step 3 Loop* ]

*Step 4.* Repeat *for* $I = 1$ to $T_Q$ by 1:

If (($C_A[I] + C_B[I] + C_C[I]$) <>

*GSABC[I]*), *then:*

Goto *Err_App_Data*.

/* Computed sum of the coefficients does not match the pre-computed sum when application data are erroneous*/

[End of If structure]

/* Compute the solutions for each quadratic equation */

Set $R_D := C_B[I]^2 - 4\, C_A[I] * C_C[I]$

$GTB2 = GTB2 + 1$

/* Increment the variable *GTB2* which is used as a check-point at this program block say, *B2* */

$GTS = GTS + 1$

/* Other check-point variables */

$GTPF2 = 2$

If $R_D > 0$ , *then:*

Set $X_1[I] := (-C_B[I] + sqrt(R_D)) / 2\, C_A[I]$

/* Solutions */

Set $X_2[I] := ( - C_B[I] - sqrt(R_D)) / 2\, C_A[I]$

Write: $X_1[I], X_2[I]$.

$GTB3 = GTB3 + 1$

/* The check-point variables at this block say, *B3* */

$GTS = GTS + 1$

$GTPF3 = 3$

Else if $R_D == 0$, *then:*

Set $X[i] := -C_B[I] / 2C_A[I]$.

$GTB4 = GTB4 + 1$

/* *Diagnostic* Check-point variables *GTB4, GTS* are incremented at this program block namely, *B4* */

$GTS = GTS + 1$

*GTPF4 = 4*

Write: "*Unique Solution", X[I].*

Else:

*GTB5 = GTB5 + 1*

*/* GTB5, GTS, GTPF5* are the diagnostic check-point variables at this program block namely, *B5 */*

*GTS = GTS + 1*

*GTPF5 = 5*

Write: *"No Real Solution".*

[End of *if* structure]

Call *EXAM-EXAMCODE*

/*A routine namely, *EXAM-EXAMCODE* is invoked in order to validate the codes of an another *EXAMCODE* procedure meant for checking the sanity of the computation. */

[End of *Step 4 Loop* ]

*Step 5*. *Exit.*

{ End of *Enhanced Algorithm-2.* }

***The Algorithm for routine EXAM –EXAMCODE:***

***Algorithm-3.*** It verifies for the sanity of codes inside the *EXAMCODE* routine. Here we need one more image of the procedure *EXAMCODE* namely, *EXAMCODE1*. It verifies all the corresponding bytes of both the images. The *EXAMCODE* routine is considered as the sanity checker or decider program for verifying the sanity of the computing application. Whereas the routine namely, *EXAM-EXAMCODE* is used for verifying whether the sanity checker or the decider program i.e., *EXAMCODE* is itself corrupted or not.

*Step1.* **Do** *byte wise comparison for both the images of the procedure EXAMCODE.*

**If** *any mismatch between two corresponding bytes,* Then:

*Reload and Restart the computation.*

/* Decider routine *EXAMCODE* is itself corrupted or victimized so initiate fail- stop kind of fault tolerance. */

**Else:**

/* if the sanity checker procedure EXAMCODE is not corrupted then call the EXAMCODE routine . */

*Call EXAMCODE*

/* Verify for the sanity of the computing application as described at Algorithm-2. */

**Endif**

{End of the Algorithm

*EXAM _EXAMCODE* }

## *The Algorithm for routine EXAMCODE:*

***Algorithm-4.*** The following steps describe how this algorithm can verify for errors or faults in the program flow or in the data memory, while an application program based on *algorithm-2,* is being executed. It refers various global checkpoint - variables namely, *GTB1,GTB3, GTB4, GTB5, GTPF1* etc., that are used in algorithm-2. It verifies for the correctness of their run – time or observed values. This procedure is invoked by the procedure namely, *EXAM-EXAMCODE.*

*Step 1*. Set *GTBI* := *GTB3* + *GTB4* + *GTB5*.

/* Compute the number of program flow through if-then-else block of algorithm-2 */

*Step 2*. If *GTS* <> $T_Q + 2I$, *then:* Goto *Err*

/* *I* is the number of iterations at step-4 of algorithm -2*/

Else if (( *GTB1* <> $T_Q$ ) .AND.

( *GTPF1* <> *1* )), *then:*

Goto *Err_B1*.

/* Check the pre-stored values of *GTB1, GTPF1* */

/* Error routine is called if computation inside the program block *B1,* is erroneous. */

Else if (( *GTB2* <> *I* ) .AND. ( *GTPF2* <> *2* )), *then:*

Goto *Err_B2.*

/* Error routine is called if computation inside the program block *B2,* is erroneous. */

Else if ((( *GTB3 > I* ) .OR. ( *GTB3 < 0* )) .OR. (( *GTB3 > 0*) .AND. ( *GTPF3 <> 3*))), *then:*

Goto *Err_B3.*

Else if ((( *GTB4 > I* ) .OR. (*GTB4 < 0* )) .OR. (( *GTB4 > 0* ) .AND. ( *GTPF4 <> 4*))), *then:*

Goto *Err_B4.*

Else if ((( *GTB5 > I* ) .OR. ( *GTB5 < 0* )) .OR. (( *GTB5 > 0* ) .AND. ( *GTPF5 <> 5*))), *then:*

Goto *Err_B5*.

Else if *GTBI <> I, then:*

Goto *Err_Control_Block.*

Else if (( *I ==10* ).AND.((*X 1 <> 0* ).OR.(*X 2 <> -2*))), then:

Goto *Err_Basic_Instructions.*

/* The *test-values* of $C_A$ , $C_B$ , $C_C$ are say, *1, 2, 0* respectively, and these values are injected onto the input data space at locations where $C_A[10] = 1$, $C_B[10] = 2$, $C_C[10] = 0$ . Therefore at *10th iteration* (i.e. for *I=10* ), the correctness of the observed values of $X_1$ , $X_2$ are validated on comparing them with their expected values.*/

Else:

Write: *"Verified OK"*

*Return.*

/*Program control returns back to *main program* i.e., to the application program (based on *algorithm-2*. )*/

[End of *If* structure]

{ End of *Algorithm-4*.}

### *Proposed Transformation Rules:*

A set of transformation rules is proposed that are applicable to the high level code. These transformations introduce data and code redundancy in the resulting program enabling detection of possible errors affecting data and code.

Rule # 1 : Every input variable *inp* must have three copies say, inp_c1, inp_c2, inp_c3

Rule #2: Run the application with the input data (with majority) to detect error in input.

Rule #3: For each program block (*i* th block), keep a counter variable *pbcn_i (global variable)* with an initialized value of 0.

Rule #4: Increment *pbcn_i* at the i th block.

Rule #5: Verify the correctness of counter variable data after the execution of a program block.

Rule #6: Inside a procedure set a global checkpoint variable to a particular value. In the calling program, verify whether that checkpoint variable contains the consistent known value. This is to check the sanity of procedure execution.

Rule # 7: Periodically run the application with known input data and then verify whether the output is the expected one. This is to verify the correctness of program code and processor.

Rule # 8: Write operation on output variable *out* should be in three-copy say, out_c1, out_c2, out_c3.

Rule #9: Final output is the value with majority on comparing the output copies. This is to mask erroneous output data.

Rule #10: After a computing session compute the checksum and compare with the previously computed and stored checksum to detect errors in program code.

Rule # 11 : If error is detected, then an Error_Routine is called in order to recover the erroneous block and/or to re-compute. Re-computation tolerates transient faults. For permanent errors, we need to recover the error affected block.

## 4. Discussion on Algorithms*:*

The *basic computing steps* for solving a series of quadratic equations are stated in *algorithm-1*. The *algorithm-2* is the *enhanced* version of the *algorithm-1*. The enhanced version is enriched with few more extra steps using *checkpoints.* The enhanced application program (as shown in *algorithm-2*) and the *EXAMCODE* routine (as shown in *algorithm-4* ) are discussed thoroughly, with necessary justifications in details. It is to examine thoroughly how this proposed technique is useful for designing a low cost fault tolerant computing application without using any extra hardware modular redundancy as the traditional *N-version programming* techniques use. The algorithm-3 verifies for any corruption inside the codes of the EXAMCODE procedure. It compares all the corresponding bytes inside both the images of the EXAMCODE. If there is any mismatch then it certainly indicates an event of code corruption and then we need to reload the application, or at least to reload or to repair the codes of the EXAMCODE. The *Algorithm-2 & 4*, are discussed below.

In *algorithm-2*, the entire program is visualized as if it has five logical *program blocks* namely, *B1, B2, B3, B4* and *B5*. *Step-1* is to read the total number of quadratic equations (i.e. $T_Q$ ) to be solved. One more variable $CT_Q$ is to store the value of $T_Q$. The *step-2* is to initialize the global variables or counters to zeroes. In other words, the check pointing variables namely , G*TB1, GTB2, GTB3, GTB4, GTB5, GTPF1, GTPF2, GTPF3, GTPF4, GTPF5, GTS* and *GTBI* are all initialized to *0*. The variable *I* is to hold the current iteration number. All these variables are *global* because these are also referred inside the procedure *EXAMCODE* (as stated in *algorithm-4* ). *EXAMCODE* is invoked inside the *alogorithm-3*. *Algorithm-4* is discussed later. Now, the *step-3* of algorithm – 2, is to read the coefficients of all the $T_Q$ number of quadratic equations. The $C_A$ coefficients of all the quadratic equations are stored in the array elements of $C_A[\ 1 : T_Q\ ]$. Similarly, the other two arrays namely, $C_B[\ 1 : T_Q\ ]$, $C_C[\ 1 : T_Q\ ]$ are to store the coefficients namely, $C_B s$ and $C_C s$ respectively. At each iteration of *step-3* loop, the counters *GTB1* and *GTS* are incremented by one. *GTB1* stands for the counter variable used inside the program block *B1*. The counter *GTPF1* is always set to *1*. The other array

*GSABC[1: $T_Q$ ]* is to store the sum of $C_A$*s,* $C_B$*s, and* $C_C$ *s*. In other words, the I th element of *GSABC* i.e. *GSABC[I]* stores the sum value of the three coefficients (i.e., $C_A$ *[I],* $C_B$ *[I],* $C_C$*[I]* ) of the *I* th quadratic equation. In an ideal case when there is no fault or error in program flow or in data memory and, after $T_Q$ number of iterations at *step-3,* the expected values of the checkpoint counters namely, G*TB1, GTS, GTPF1* are equal to $T_Q$ , $T_Q$ , *1* respectively. At each iteration of *step-3,* the value of the counter *I* is validated on comparing with the memory content of the variable $CT_Q$ . If *I* is greater than $CT_Q$ , it is obvious that either the counter *I* or the variable $CT_Q$ is corrupted. Again, there is a possibility of corruption at the *repeat* instruction. Thus the "*if"* conditional statement at *step-3*, can detect such possible data corruption and then it can invoke an error routine namely, *ERR_B1*.

At *step-4*, the values of $R_D$ *s (i.e.,* $C_B[I]^2 - 4\ C_A[I] * C_C[I]$ *)* and, the solutions *i.e.,* $X_{1s}$, $X_{2s}$ are computed for all the quadratic equations on executing $T_Q$ number of iterations. At each iteration say, at the *I* th iteration the sum of $C_A$ *[I],* $C_B$ *[I],* $C_C$*[I]* is computed and it is compared with the pre-computed value of *GSABC[I]*. If there is a match, the program flow goes forward to compute the $R_D$ *s ,* $X_1$, $X_2$. Otherwise, an error routine namely *ERR_App_Data* is invoked. In other words, before computing the solutions of the *I* th equation, the application data are validated. If one of the application data is corrupted then the sum of the *coefficients* will not match with the pre-computed sum value namely, *GSABC[I]*. The corrupted application data will definitely lead to wrong results. Again, at the "*If" statement* at *step-4* of *algorithm-2*, it is obvious that if the application data are corrupted then the *I* th element of the pre- computed ( at *step-3* of *algorithm-2* ) *GSABC[I]* will not match with the recomputed sum of $C_A$ *[I],* $C_B$ *[I], and* $C_C$ *[I]* . In other words, $(C_A[I] + C_B[I] + C_C[I]) <> GSABC[I]$. As the events of data - corruptions are *mutually exclusive,* the chances that the corrupted data elements (each of *32-bit),* in the arrays namely, $C_A$ *[I],* $C_B$ *[I],* $C_C$ *[I] , GSABC[I]*, will satisfy the relation at *step-4* , is stated below.

The conditional probability of an error is 0 if only one of the variables in the comparison is corrupted and it is (1/2)**32 if two or more are corrupted in arbitrary ways.

Thus, the verification of application data as performed by the *"If" statement* at *step-4*, is carried out correctly because the *failure probability* for detecting the corrupted application data at this *checkpoint*, is *zero* (as shown above). So, the corrupted application data are detected at this point of execution time. When data corruption is detected, an error-routine namely, *Err_App_Data* is called for taking necessary recovery action like copying the application data from its master copy and re-executing the application. At each iteration of the step-4 loop, we detect errors in application data. It prevents error propagation in order to avoid disastrous consequences of the application program. At each iteration, the counters *GTB2*, *GTS* are also incremented by one inside the program block *B2*.

Again, depending on the value of $R_D$ , only one among the three program blocks namely, *B3*, *B4* and *B5,* is executed at each iteration. Thus, only one among the three counters namely, *GTB3, GTB4, GTB5,* is incremented at each iteration. So, at the end of *I* th iteration of *step-4*, the expected values of all these flags or the counters namely *GTB2, GTS, GTPF2, GTPF3, GTPF4* and *GTPF5* are *I,* $T_Q + I$, *2, (0* or *3)*, (*0* or *4*) and (*0* or *5*) respectively. For an example, when the values of $R_D s$ are greater than *0,* then the program flow does not enter at all inside the program block *B3* . Therefore, the expected value of *GTPF3* is *0.* Otherwise it is *3*. At the end of *I* th iteration, the expected value of the sum of those three counters ( i.e. *GTB3 + GTB4 + GTB5* ) is *I* only.

The *expected value of a variable or flag is the value of a variable when there has been no occurrence of faults or errors in the program flow and data memory.* At the end of each iteration say, at *I th* iteration, and before starting the next iteration (i.e. *I+1* th iteration), the error or fault checking routine namely, *EXAM-EXAMCODE* ( as stated in *algorithm-3* ) is invoked which in turn calls another procedure namely, *EXAMCODE* (as shown in algorithm-4) in order to validate the memory contents of those variables, counters or flags. In other words, at the end of each iteration of *step-4* of *algorithm-2*, the *EXAMCODE* routine is

executed through the *EXAM-EXAMCODE* routine in order to verify whether there is any possible discrepancy between the *expected* and *run time* (or observed) values of a checkpoint variable as mentioned above. *The discrepancy between the expected and run time values does exist when there occurs an abnormal program flow or the unintentional alteration of the stored data.* This discrepancy between the expected (or theoretical) and observed (or practical) values of a variable is not uncommon because the processor of today's computer, with high frequency clock, that runs applications at lightning speed is often victimized by the potential electrical fast transients or transients. As a result, random errors may occur within the processor systems. The situation is aggravated by a noisy industrial environment. The procedure namely, *EXAM-EXAMCODE* verifies for faults inside the codes of the procedure namely *EXAMCODE* . The sanity of the EXAM-EXAMCODE may also be verified by using some form of checksums.

## 5. The Semantic Based Assertive EXAMCODE Routine

Depending upon the application program and data flow semantics, we derive the various assertions based diagnostic checkpoints and we need to carry out run-time examination of those assertions based checks to understand the cause of errors. The *step-1* of the *algorithm-4* is to store the sum of three diagnostic checkpoint variables namely *GTB3, GTB4* and *GTB5* on to a variable *GTBI. Step-2* is to validate the memory contents of those variables. Now, at *I* th iteration of *step-4* loop of the *algorithm-2* (i.e. just before invoking the *EXAM-EXAMCODE* routine), the expected value of the variable *GTS* is $( T_Q + 2I )$. The expected value of *GTS* is $T_Q$ at the end of $T_Q$ number of iterations at *step-3* of *algorithm-2* . Again, the variable *GTS* is incremented by two at each iteration at *step-4* of *algorithm-2.* So at the end of the *I th* iteration, the expected value of *GTS* is $( T_Q + 2I )$. The " *if "* statement at *step-2* of *algorithm-4*, verifies for the presence of any discrepancy between the expected and the observed values of *GTS*. Again, at *I th* iteration, if the observed value of the variable *GTS* is not equal to the expected value i.e., $( T_Q + 2I )$ , then it is obvious that either the stored values of those variables are

corrupted or, there has been an error in the program flow while executing the instructions at *step-3* through *step-4* of the *algorithm-2*. Thus if such errors are present, they are detected at each iteration and the program flow is brought to an error routine *ERR* for executing it. Similarly the first *" else if "* statement of the *step-2* of the *EXAMCODE routine* verifies the correctness of the values of *GTB1,* $T_Q$ *,* and *GTPF1* variables. The expected values of *GTB1 , GTPF1* are equal to $T_Q$ and *1* respectively. If *GTB1* is not equal to $T_Q$ , it is obvious that the loop, at *step-3* of *algorithm-2,* is not completed correctly due to faulty program flow or due to corrupted values of $T_Q$ *, GTPF1 and GTB1.* Thus it indicates the presence of faulty program flow or errors in the data memory at the program block *B1*. Again, the expected values of the checkpoint variables namely, *GTB2,GTB3*, *GTB4, GTB5, GTPF2, GTPF3, GTPF4 and GTPF5* are verified at the subsequent *"else if"* statements at *step-2* of *algorithm-4*. These *"else if"* statements are meant for verifying the sanity of other program blocks. For an example, inside the *block B4* of *algorithm-2*, the expected values of *GTB4* and *GTPF4* are as follows.

$0 <= GTB4 <= I$ , and,

$GTPF4 = 4$ for $GTB4 > 0$. *(1)*

Similarly, inside the program *block B5* of *algorithm-2*, the expected values are:

$0 <= GTB5 <= I$, and,

$GTPF5 = 5$ for $GTB5 > 0$.*(2)*

The expected value of *GTBI* is *I* at the control block *'if- else if- else'* at *B3, B4 and B5.* Because the sum *(GTB3 + GTB4 + GTB5)* is incremented by *1* at each iteration at *step-4* of *algorithm-2* and it depends on the value of $R_D$ . In other words, at each iteration, when *GTB3* is incremented by one, the other two counters *GTB4, GTB5* are not incremented and so on. Because the transients or electromagnetic pulses cause the inadvertent alterations of the stored bit patterns randomly, therefore the various conditional statements inside the *If - then- else*

conditional blocks, in the *EXAMCODE* routine, validate all the stored information accurately. For an example, if the various counters or variables are say, 32 bit integers, then the possibility of producing wrong decision based on such various conditional statements is *nil*. For an example, at the first *if* statement at *step-2* inside the *EXAMCODE* routine, if the values of the variables namely, *GTS,* $T_Q$*, I* are corrupted and, then such corrupted values can not satisfy the equation $GTS = T_Q + 2I$. Thus, it prevents initiation of false action caused by producing wrong decision. For an example, if after *I - th* iteration at *step-4* of *algorithm-2, the expected values* of the variables *GTS*, $T_Q$, *I* are say, $GTS_{exp}$, $T_{Q\,exp}$, and *I*, then these expected values will obviously satisfy the following equation.

$GTS_{exp} = T_{Q\ exp} + 2\,I_{exp}$ . *(3)*

## ***Errors in Program Flow*:**

Assume that an event such as a transient has caused some faults or errors in the program flow ( at *step3, step4* of *algorithm-2*), and as a result, the values of *GTS,* $T_Q$*, I* get altered. Let $GTS_{pfe}$, $T_{Qpfe}$, and $I_{pfe}$ be the *observed values* of these variables after the error-producing event. The event is detected, with high probability, since

$GTS_{pfe} <> T_{Q\,pfe} + 2\,I_{pfe}$ . *(4)*

*The Data Errors:* Assume that an event such as transient has caused some data errors of the application program, changing the values of *GTS,* $T_Q$, and *I.* Let $GTS_{de}$, $T_{Qde}$, and $I_{de}$ be the *observed values* of these variables after the error-producing event. The event is detected, with high probability, since

$GTS_{de} <> T_{Q\ de} + 2\,I_{de}$. *(5)*

The Transients cause the data corruption completely in a random fashion. If we consider the integer variables each of say, 32 bit, then the chances of *GTS,* $T_Q$ and *I* being corrupted to a combination of values such that $GTS_{de} = T_{Q\ de} + 2\,I_{de}$ is negligible, with probability

$$( 1 / 2^{32} ) * ( 1 / 2^{32} ) * ( 1 / 2^{32} ) = 1 / 2^{96} \approx 0.$$

## ***The Recovery Works***

The *EXAMCODE routine* detects and initiates recovery actions. The errors or faults in the program flow as well as the errors in the application data area, are detected by the sanity checker or the decider routine namely, *EXAMCODE* and then the *error routines* namely, *Err, Err_B1* etc., are invoked for the necessary recovery actions e.g., the immediate termination of the program execution and switching to the recovery routines. At each *program block* of *algorithm-2,* an appropriate error routine is invoked whenever errors are detected by the *EXAMCODE* routine. Examples include the following.

- The *Err_B2* routine is invoked when errors or faults occur in the program *block B2* of the *algorithm-2*.
- The *Err* routine is invoked in the case of an error while computing the *I* - th quadratic equation (i.e. for an error during the *I*-th iteration of *step-3* and *step-4* of the *algorithm-2* ).
- The *Err_Control_Block* routine is invoked in response to an error in the "if-then-else" of *step-4* of *algorithm-2.* Inside the program block, the sum *GTB3 + GTB4 + GTB5* is incremented by *one at each iteration.* Depending on the location (as detected by the *EXAMCODE* routine) and nature of corruption or faults, the programmer may repair the corresponding corrupted code or, may restore the application program from a master copy on floppy or magnetic tape and re-execute the program.

An *error routine* relates the presence of errors to the corresponding iteration – number and to the program block name. For example, the invocation of the *Err_B4* routine indicates the value of *I* and the name of the contaminated program block *(B4).* Any inadvertent jump of the program flow is detected by means of different conditional statements of the *EXAMCODE* routine. For example, $GTB2 \neq I$ at *step-2* indicates a program flow error. An infinite looping is also prevented (e.g. the *step-4 loop* ) by the *EXAMCODE* routine. The

unused memory area is filled with *"No Operation"* codes terminated by a *Return* statement. This prevents the infinite loops often caused by an inadvertent jump of the program flow to the unused memory.

Recovery actions can be defined relatively easily since all the error – routines are designed to include the iteration number *I* (when the error has occurred) and the program block name (where the error has occurred ) . If some of the instruction codes are corrupted, then there will be an error in the program flow and the counters and variables used at various check points may contain incorrect values. This *mismatch* between the expected and observed values of the various counters, variables, flags is detected at various *"If – Then – Else"* conditions at *step-2* of *algorithm-4.* These *error –routines* repair the data and application program, and then in order to initiate re-execution of program, they will need to address the erroneous values of these error detection variables, counters, checkpoints and flags.

*Examples of such corruption* of error detection variables are as follows.

• If the instruction code corresponding to *GTB1 = GTB1 + 1* at *step-3* of *algorithm– 2*, is corrupted, then the counter *GTB1* is not incremented at each iteration and thus the relation $GTB1 <> T_Q$ at the first *Else-If* statement of *step-2* of the *EXAMCODE* routine will be true.

• False branching (i.e., a faulty program flow ) may cause skipping of the execution of an uncorrupted instruction, e.g., *GTB2 = GTB2 + 1,* at *step-4 loop.* This false branching will lead to an erroneous value of *GTB2*.

• A wrong reset of a variable (e.g., *GTB1* ) will result in a discrepancy between the variable's expected and observed values. This discrepancy is detected effectively by the *EXAMCODE* routine. At each iteration of *step-4,* the *EXAMCODE* routine is invoked via a procedure namely, *EXAM-EXAMCODE*. At the *I* th iteration, the expected value of say *GTB2* is *I* and, in the second *"Else-If"* statement at *step-2* of the *EXAMCODE* routine, the relation *GTB2 .NE. I* is satisfied if *GTB2* is reset wrongly leading to the error routine *Err_B2* being invoked. In addition to using values of variables, flags, and counters established by program execution, one may also *"inject"* values into

variables in order to externally monitor the *correct progression* of the program. For example, at a given iteration $I$ , one may validate the computed values of $X_1$ and $X_2$ with respect to their expected values by externally loading relevant variables with their expected values. Any discrepancy between these expected and observed values of $X_1$ and $X_2$ for predefined known values of $C_A$, $C_B$ and $C_C$ can then be detected. For example, one can inject one set of test input data ( e.g., values of *1, 2,* and *0* for $C_A$ , $C_B$ and $C_C$ , respectively) into the input data arrays at the $10^{th}$ element establishing $C_A[10] = 1$ , $C_B[10] = 2$, and $C_C[10] = 0$. With these values of the coefficients, the expected values of $X_1$ and $X_2$ are 0 and –2, respectively. If the *observed values* do not match these expected values , the basic instructions might have been corrupted, and in that case , the error routine *"Err_Basic_Instructions"* (in the *EXAMCODE* routine) is invoked. One may *inject test values of coefficients* at several locations (say $I$ = 10, 20, 30 …) of the input arrays. It is not necessary to save the corresponding computed results for the test-input data in an output data file. If those faults of transient nature occur, the execution of the application program is terminated and again re-executed.

Since the corresponding computed results for the test-input data may not have been saved in an output data file, for those faults of transient natures, the execution of the application program is terminated and again re- executed.

The basic instructions and the additional instructions have the same fault susceptibility, given the randomness of corruptions due to transients. *The technique described here checks both the basic instructions and the additional instructions used to provide fault tolerance.* The checking capabilities include both *automatic checking during program execution and external testing by injection of test variables and insertion of checkpoints.*

In summary, if there is an error or fault in the program flow or in the data memory

(caused by short duration noise pulses such as *Electrical Fast Transient, Transient* pulses etc.), the *EXAMCODE routine* will detect the error or fault immediately. The variables (*GTB1, …, GTB5*), (*GTPF1, …, GTPF5*), *GTBI* and

*GTS* are used to detect the faults or errors in data and program flow. These variables are used to provide *"diagnostic check points"* inside the application program. If an error has occurred, the necessary recovery action is taken immediately to prevent propagation of errors to the next iteration and to avoid further damage in the application. This technique is also very useful for locating various faults, if occurred, by means of those *checkpoints* ( i.e., the iteration *I* and the program block at which the error occurred). *Transient – contaminated blocks of the application program are easily identified and restoring the contaminated block with a backup copy can be initiated.* The novelty of this approach is its inherent easiness and simplicity of its implementation without using traditional *N-Version* programming technique. *The affected program block may also be repaired* on keeping three images of the application code and data [11]. Again the sanity of the codes of the decider or examiner routine namely EXAMCODE is also verified by another routine namely, EXAM-EXAMCODE. The memory overhead is with the extra codes in computing algorithm – 2, the code of the EXAM-EXAMCODE and two images of the decider routine namely, EXAMCODE. Because of the present trend of lowering the cost of memories, a system designer can easily afford such space redundancy. Again, the execution time overhead can also be ignored because of the affordable high-speed modern computing machine. It has been observed that execution time overhead is on an average of the order of 1.52 times the basic logic and the executable code size grows to 1.6 times the basic logic without any software fix. It is observed that 98.8% of various errors that we injected manually at the parts of checkpoints and at loops in the source code have been detected by this single-version algorithmic approach.

## 6. Conclusions

The method of such low cost semantic based assertive *software fix*, as demonstrated in this paper, has been developed in order to provide a *cost effective solution* to *fault tolerant computation* in the presence of electrical noises causing contamination of the program execution. The approach is a software

approach, not requiring redundancy of hardware but also not requiring *N*-version programs because, this single-version approach aims to tolerate run-time operational errors, not the software design bugs . *This approach establishes test conditions within a single version of the software program, rather than requiring multiple and distinct versions of the same "application program" running concurrently on many machines.* However, a thorough knowledge of the application system is necessary for designing such enhanced processing logic illustrated here by the example of solutions to sets of quadratic equations. System designer or programmer has to use thoughtful checkpoints at proper places. With practices one can find it easy to design such single – version software. Again, for more fault coverage system designer can use more checkpoints and may use another image of the application program also. In other words, depending on the reliability and speed of the processing unit, the imposed specifications for the reliability of an application (e.g., allowed probability of not detecting an error ), and the response time and frequency of fault occurrences caused by noises , a number of variables , flags and checkpoints can be adjusted to best satisfy requirements. In the case of *high frequency of faults* (e.g., Transient faults at a thermal power plant or at a heavy engineering industrial area), the *number of required checkpoints* is higher than for conventional applications in which fault frequencies are substantially lower. The method described here using enhanced software logic is presented as a useful and cost effective software technique for incorporating fault tolerance into computational systems. It avoids the overhead on the hardware cost with the conventional high modular redundancy, though it must be recognized that the approach addresses only transient faults due to external electrical noise pulses corrupting data or digital logic operation in a transient , rather than permanent manner. The software solution does require some additional execution time without using multiple versions of applications. The fault checking procedures can also be substituted by macros for faster execution. *A superficial knowledge of the system and its operating environment may produce a fault tolerance solution, which is substantially, sub optimal* . However,

several applications can exploit the very high speed of contemporary processing units and memory to relax the constraints of achieving optimal designs for a given application. This approach is also useful for various computer-controlled applications.

***Further Readings:***

[1] A. Avizienis, "The N - Version Approach to Fault Tolerant Software," IEEE *Transactions on Software Engineering,* vol. SE-11, December 1985, pp. 1491-1501,.

[2] Goutam K Saha, "EMP -Fault Tolerant Computing: A New Approach," *Journal* of Microelectronic Systems Integration, Vol. 5, Number 3, 1997, pp.183 - 193, Plenum Press, USA .

[3] Y.N. Shen, H. Kari, S.S. Kim, and F. Lombardi, "Scheduling Policies for Fault Tolerance in a VLSI Processor," in *Proc.IEEE* Int. Workshop on DFT in VLSI Systems, Montreal, Canada, October 1994, pp. 1-9.

[4] J.W.S. Liu, W.K. Shih, K.J. Lin, R. Bettati, and J.Y. Chung, "Imprecise Computations," *Proc. IEEE*, vol. 82, no.1, Jan. 1994, pp.83-94 .

[5] B. Randell, et al., "Reliability in Computing System Design," *ACM Computing Surveys,* Vol. 10, Number 2, 1985, pp. 374 –382.

[6] Goutam Kumar Saha, "Low-Cost, Fault Tolerance Applications," IEEE *Potentials*, vol. 24, no. 4, November 2005, pp. 35-39.

[7] M. Zenha Rela, H. Madeira, J.G. Silva, "Experimental Evaluation of the Fail-Silent Behavior in Programs with Consistency Checks", Proc. FTCS-26, 1996, pp.394-403.

[8] K.H. Huang, J.A. Abraham, "Algorithm - Based Fault Tolerance for Matrix Operations," IEEE Trans Computers, Vol 33, Dec 1984, pp. 518-528.

[9] S.Yau, F. Chen, " An Approach to Concurrent Flow Checking," IEEE Trans on Software Engineering, Vol. SE-6, No. 2, March 1980, pp. 126-137.

[10] D.K. Pradhan, "Fault-Tolerant Computer System Design", Prentice Hall PTR, 1996.

[11] Goutam Kumar Saha, "Software Implemented Fault Tolerance Through Data Error Recovery," ACM Ubiquity, Vol. 6 (35), ACM Press, USA, 2005.
[12] Goutam Kumar Saha, "Transient Fault Tolerance in Mobile Agent Based Computing," INFOCOMP Journal of Computer Science, Vol. 4, no.4, UFLA, Brazil, 2005, pp. 1-11.
[13] Goutam Kumar Saha, "Fault Tolerance in Web Services," ACM Ubiquity, Vol. 7 (9), ACM Press, USA, 2006.
[14] Goutam Kumar Saha, "Software Based Fault Tolerant Computing," ACM Ubiquity, Vol. 6 (40), ACM Press, USA, 2005.
[15] Goutam Kumar Saha, "Transient Software Fault Tolerance Using Single-Version Algorithm," ACM Ubiquity, Vol. 6 (28), ACM Press, USA, 2005.
[16] Goutam Kumar Saha, "Transient Fault Tolerance Through Recovery," ACM Ubiquity, Vol. 6 (35), ACM Press, USA, 2005.
[17] Goutam Kumar Saha, "A Software Fix Towards Fault Tolerant Computing," ACM Ubiquity, Vol. 6 (16), ACM Press, USA, 2005.
[18] Goutam Kumar Saha, "Software Based Fault Tolerant Array," IEEE Potentials, Vol. 25 (1), IEEE Press, USA, 2006.
[19] Goutam Kumar Saha, "Transient Fault Tolerance Through Algorithms," to appear in IEEE Potentials, USA, 2006.
[20] D.W. Bradley, C. Ortega-Sanchez and A.M. Tyrrell, "Embryonics + Imunotronics: A Bio-Inspired Approach to Fault Tolerance," In Proceedings of 2nd NASA/DoD Workshop on Evolvable Hardware, July 2000.
[21] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthi, J.A. Abraham, "Design and Evaluation of System - Level Checks for On-line Control Flow Error Detection, IEEE Transactions on Parallel and Distributed Systems, Vol.10 (6), Jun 1999, pp. 627-641.
[22] R. Reddy, R. France,G. George, "An Aspect Oriented Approach to Analyzing Dependability Features," Proceedings of the AOM-AOSD, 2005.

**Author's Biography:**

Goutam Kumar Saha (gksaha@rediffmail.com, sahagk@gmail.com) has been working as a Computer Scientist for last eighteen years. He has worked in various renowned research organizations namely at *LRDE*, Defence Research & Development Organisation *(DRDO)*, Bangalore, at Electronics Research & Development Centre of India *(ER&DCI)* Calcutta. His present employer is Centre for Development of Advanced Computing *(CDAC),* Kolkata. He is a Scientist-F in CDAC. He is a senior member in IEEE (USA), Computer Society of India (CSI). He is a Fellow Member in IETE, MSPI (New Delhi), IMS (Goa). He received various grants & awards from foreign and national reputed institutions. He has authored around one hundred research papers on fault tolerant computing, dependable computing and NLP. He is a referee of the CSI Journal, AMSE Modeling Journal (France), IJCPOL (USA) and an IEEE - Magazine. He is an associate editor of the ACM Ubiquity. He is also an active member of the WWW Consortium Internationalization Tag Set Working Group (W3C-ITS-WG).

Ubiquity Volume 7, Issue 22 (June 13 - June 19, 2006)

http://www.acm.org/ubiquity/