Evaluation of important reliability parameters using VHDL-RTL modelling and information flow approach

Mehdi Jallouli, Hicham Belhadaoui, Camille Diou, Fabrice Monteiro, Olaf Malassé, Jean-François Aubry, Abbas Dandache, Grégory Buchheit, Hicham Medromi

Evaluation of Important Reliability Parameters using VHDL-RTL modelling and Information Flow Approach

Mehdi JALLOULI[1], Hicham BELHADAOUI[2][4], Camille DIOU[1], Fabrice MONTEIRO[1], Olaf MALASSE[2], Jean-François AUBRY[3], Abbas DANDACHE[1], Grégory BUCHHEIT[2] and Hicham MEDROMI[4].

[1] *: Laboratoire Interfaces, Capteurs et Microélectronique, Université Paul Verlaine 7 rue Marconi, 57070 Metz, France*
[2] : *A3SI, ENSAM, A&M ParisTech, 4 Rue Augustin Fresnel 57078 Metz Cedex 3, France*
[3] : *CRAN, Nancy Université, CNRS, 2, Avenue de la forêt de Haye 54 516 Vandœuvre-lès-Nancy Cedex, France*
[4] : *Equipe Architecture des Systèmes, ENSEM, Route d'El Jadida, km 7 BP. 8118 Oasis Casablanca, Maroc*

Abstract

Fault tolerance is an essential requirement for critical programming systems, due to potential catastrophic consequences of faults. Several approaches to evaluate system reliability parameters exist today; however, their work is based on the assumptions that hardware and software failures happen independently. The challenge in this field is to take into account the hardware-software interactions in the evaluation of the model.

In the continuity of the CETIM project [1] whose principal objective is to define an integrated design of dependable mechatronic systems, this work evaluates important reliability parameters of an embedded application in a stack processor architecture using two dynamic models. The first one (stack processor emulator [2]) allows the study of dynamic performance and the evaluation of a fault-tolerant technique. The second one (information flow approach [3]) evaluates the failure probability for each assembler instruction and for some program loops. The main objective is to estimate the failure probability of the whole application. The hierarchically modelling with the information flow approach makes it possible to evaluate the efficiency of protection program loops. These loops ensure the fault tolerance policy by recovering imminent failures and allow the application to run successfully thanks to a permanent software recover mechanism: in case of a detected and not corrected error, the system returns to the last faultless state.

This work is useful because it allows adjusting the architecture and shows the advantages of the hardware-software interactions during the co-design phase before the hardware implementation. It puts the hand on the critical points in term of reliability thanks to the scenarios of critical failure paths in the processor architecture.

Keywords *- embedded systems, device modeling, reliability issues, VHDL-RTL modeling, fault-tolerant, IEC61508, stack processor.*

1. Introduction

Computer systems operating in industrial environment are subject to different radiation phenomena, whose effects are often called '*soft error*'. Generally, these systems employ software techniques to address and tolerate the soft errors. In this paper, a software fault tolerance technique based on a recovery strategy is evaluated using information flow approach. The distinctive advantage of this fault tolerance technique is the possibility to estimate the time performance overhead depending on the fault apparition. By applying the information flow modeling on several benchmark applications, we evaluate the probabilities of existence on different functional mode.

Most of the studies, however, have focused on fault coverage and error latency of hardware fault-tolerant mechanisms in digital systems as dependability measures [4]. At recent years, it was reported that the environmental transient faults could be masked only by software without hardware error masking mechanisms [5]. Thus, a substantial number of faults do not affect the program results for several reasons: faults whose errors are neutralized by the next instructions, faults affecting the execution of instructions that do not contribute to the benchmark results, and faults whose errors are tolerated by the semantic of the running benchmark. This effect should be considered properly because even a small change of system fault coverage value can affect the system dependability [6].

Using the stack processor emulator and benchmarks, an emulation model for fault injection was developed to estimate the dependability of the complex programmable system in operational phase. Through the fault injection into the emulator, the software fault tolerance effect on system is evaluated in this work. This effect is measured by the flow

informational modeling approach. The faults consist of the single bit-flip, hardware fault (failure of component) and stuck-at-value faults in the internal registers of the stack processor and in memory cells.

We start by introducing the faults tolerance strategy proposed by LICM laboratory in Section 2. This strategy is described and explained how it can used efficiently. In Section 3 the information flow approach is introduced in details. It's firstly applied on one instruction before being generalizing on the whole bubble sort program. The probabilities of existence on different functional are calculated. Section 4 gives assessment of the same probabilities but with taking into account the fault tolerance strategy. These results are analysed and compared with Section 3's results. Finally, Section 5 concludes and presents the future works.

2. Fault tolerance strategy

The protection techniques consist of a share between hardware techniques and software techniques (Fig 1). Here, we interest to errors which can be produced in the processor and not in the application. If we look at the consequences of these techniques, we notice that on one hand, when implementing hardware techniques, we will have an additional silicon area and possibly extend the critical path. On the other hand, when implementing software techniques, we will have a loss on time performance because of the additional protection routines which can extend the duration of the whole program.

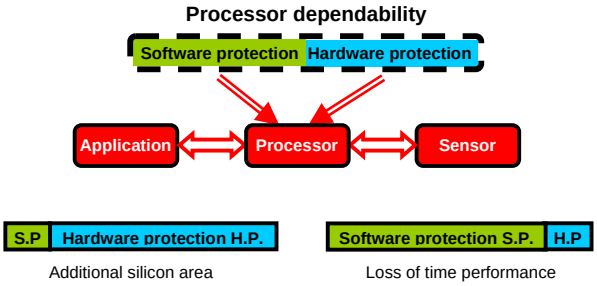

Fig 1. Necessity of compromise between software and hardware protection techniques

So, for an application located in particular conditions (error rate, time constraints, power constraints), we can estimate which kind of protection shall we use. Thus, a compromise should be found. In summary, each protection technique has its efficiency and cost. For that reason, we use both emulator and benchmark to determine the limit between software and hardware protections.

Some functional errors can be modelled and integrated in the processor emulator. As a consequence, the embedded program and the internal processor states may change. In order to skip such case, some protection techniques can be added in the benchmarks as it is shown in Fig 1. This is very useful because our priority is the dependability.

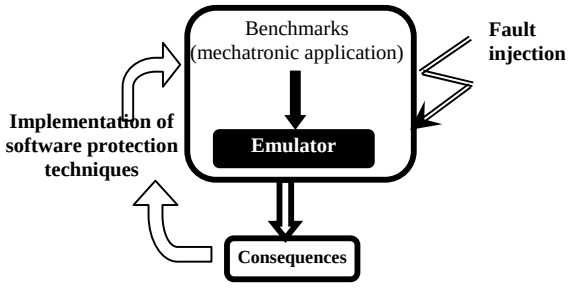

Fig 2. Usefulness of the emulator/benchmark for fault injection and protection methods integration

The errors injection is implemented in the emulator. We suppose the existence of mechanism of hardware detection technique. Each detected and not corrected error generates an interruption. This interruption forces the processor to run a routine from a definite address. The function of this routine is to return to the last dependable state. In fact, in the benchmark, we integrate a routine which store periodically the stack pointers, the program counter and the top-of-stacks. Storage of the treated data is also possible. So, on every interruption, the protection routine recovers the last data saved. These 2 routines (storing data and recovering stored data) are implemented in the benchmark. Some additional instructions are proposed in order to allow the storage of stack pointers value in the top-of-stack.

Since the fault tolerance strategy is well described, we can now introduce the global reliability evaluation strategy.

2.1 Global reliability evaluation strategy

The reliability of software is defined according to ISO/IEC 9126 [7] as '*the ability of the software to maintain a level of required performance when it is used under specified conditions*'. To avoid confusion between the hardware reliability and software reliability, this paper treats the reliability evaluation of the hardware architecture from a case study of one instruction. It will be generalized on the whole architecture instruction set. We consider that software (embedded program) is largely deterministic without taking into account its random reliability [8]. Fig 3 introduces the global reliability evaluation strategy.

Based on the standards specifications description, the original HW/SW architecture is proposed and its instruction set is designed. Each instruction is modelled by a VHDL-RTL model to facilitate the transition to its information flow stochastic model, from which the high-level and low-level with eventual simplification are made. An emulator is developed based on the instruction set. Obviously, benchmarks are developed to test them on this emulator. These benchmarks represent mechatronic applications that can be embedded in the stack processor in the future. Because dependability is our main goal; an injection of faults

is integrated into the emulator, periodic, random, or salve injection. Through the different fault injection into the emulation model, the software fault tolerance effect is studied: the impacts on time performance and the ability to correct faults in different scenarios of errors occurrence. Another parameter to evaluate this software fault tolerance method is based on the informational flow modeling approach. The software fault tolerance method is integrated in the informational flow of each instruction in order to have a quantitative probabilistic assessment. Using this approach, we predicate the software fault tolerance coverage values in a programmable system and estimate this reliability. We obtain a global analysis which may judge this protection method and which serve to an eventual architecture fine-tuning.

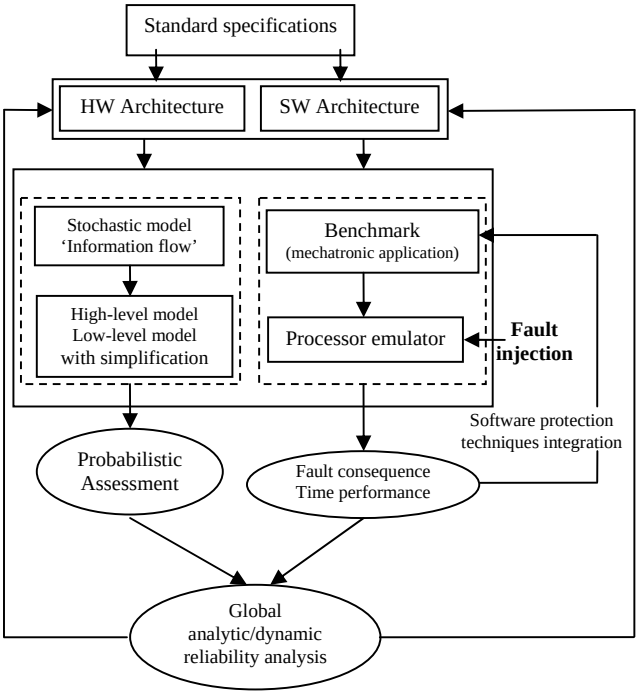

Fig 3. Co-design flow using a global reliability evaluation strategy

So, we apply this modeling on one assembler instruction based on its RTL modeling. This part is detailed in the next section.

3. Information Flow approach applied on sorting program (without tolerance)

3.1 Case study

The case study is the bubble sort of variables. From a table containing these variables, the program consists of reading the data, swapping in the case of disorder and kipping the table with new ordered values. First of all, the program initializes the variables in memory. We make a reset of the permutation flag. Every two successive data is compared. If a permutation is done, the flag is putted to 1. We make the same comparison until the end of variables and until the flag is null.

To evaluate the global probability of the correct run of the program after sorting the variables, we are placed in the worst case where these variables are totally disordered. The following figure shows that in reality the software application is composed of blocs. The functions and routines composing these blocks are constructed by several assembler instructions. The number of such instructions changes according to the application's program.

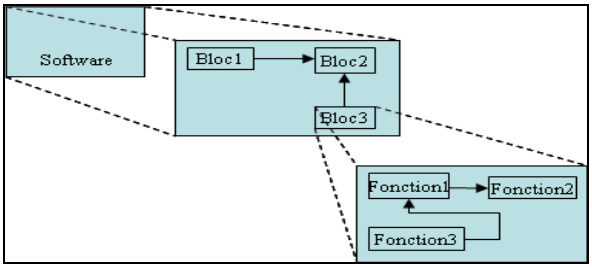

Fig 4. Decomposition of the software application

Based on the results from the probability to run correctly the separate instructions, this work shows the practicability of the information flow approach for the evaluation of program formed by several instructions, the result of this step is useful when we want to validate the effectiveness of the fault tolerance strategy proposed, as well as to validate our modeling of hardware-software interactions. In this step, it's necessary to show how we find the different values of probability for simple instruction. It's detailed in the next paragraph.

3.2 Evaluation of the simple instruction

In this part, a preliminary VHDL-RTL modeling is done for all assembler instructions in order to make easy their information flow modeling. We give one sample: the DUP instruction. Before explaining these instructions, we should mention that the processor has two stacks. One stack is used for the data treatment called data stack (DS). The top-of-stack (TOS) and the next-of-stack (NOS) correspond respectively to the first and the second element of this stack. The second stack is used for the subroutine return addresses, interruption addresses and temporary data copies, and is called return stack (RS). The top-of-return-stack (TORS) corresponds to the first element of this stack. The stack buffers are managed in an external memory of the processor in order to have no restriction in the stack depth. They are addressed by internal pointers (data stack pointer DSP and return stack pointer RSP).

Since some architecture features are explained, we can detail clearly the DUP instruction. The DUP instruction allows the duplication of the top-of-stack (TOS).

- DSP ← DSP +1 ***« incrementation of DSP to push a new element »***
- 3rd DS Element (Memory value addressed by the new DSP) ← NOS ***« NOS becomes the 3rd DS element »***
- *NOS ← TOS* ***« duplication of the TOS value »***

Fig 5 presents the RTL model of the DUP instruction.

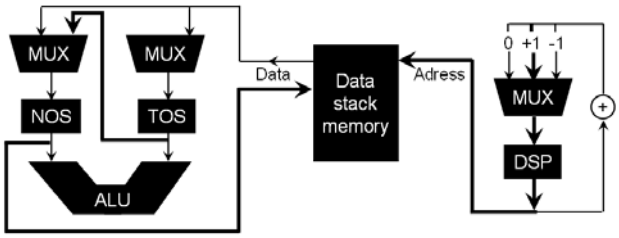

Fig 5. RTL model of the DUP instruction

Concerning the control logic for the execution of this instruction (bold lines in Fig 5):

- The data stack memory is in write enable mode;
- The control signals of the Mux_DSP are in position so that the (+1) input is selected;
- The control signals of the Mux_NOS are in position so that the connection from TOS is selected.

At the clock's rising edge, all the operations are done and DUP instruction is executed.

Thanks to this VHDL-RTL model of the DUP instruction, we can now realize the information flow modeling for this instruction. However, we should summarize the main features of this approach before [9, 10 and 11]. It is based on the following points:

- Existence of information flow;
- Existence of control events;
- Existence of check procedures;
- Existence of information storage entities.

The high level model allows an efficient subdivision of the functional architecture. The incorrect information existing in a sub-functional entity is propagated in the following ones. If this incorrect information is used by two distinct downstream sub-functional entities, it must be propagated logically along two different paths. We distinguish in high level model five types of sub-functional entities:

- TF: Entity of signal transformation: this bloc transforms the input signal into a different output signal;
- SB: Entity of signal storage: it conserves information and allows the storage of an input signal in order to be reused by another entity. The SB bloc allows representation of the necessary storage processes;
- IP: Decision entity: it allows delivering on output information with taking into account two input signals;
- CT: Entity of time control: it checks the information reception for each interval of time;
- ST: Self-test entity: it corresponds to the real time operation tests, representing on-line tests based on the observation of periodical transmission.

The result of the application of the high-level model on the DUP instruction is illustrated in Fig 6.

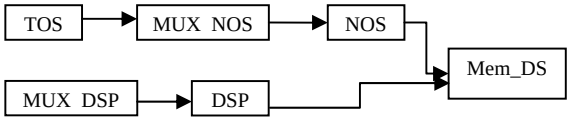

Fig 6. High-level model of DUP instruction

The DUP instruction copies the top of stack (TOS) into the next of stack register (NOS) and pushes the NOS to the third element in the data stack memory (Mem_DS). This instruction needs to increments the data stack pointer (DSP) to allocate a new cell in Mem_DS.

Concerning the low-level model, it consists of modelling the functional and dysfunctional behaviour of each sub-functional entity in the high level model by associating a finite state automaton [12]. The transition events in this automaton are classed and labelled between different states. The transient failures (bit-flip), which can change the credibility of information, are mainly modelled by associating them with random events. The hardware failure modes (dM) related to the sub-functional entity are also modelled, we talk about permanent failure modes. It is possible to model the following modes: non-achievable state, correct behaviour, dead-lock situation, infinite loop, live-lock situation and the forbidden types of communication between the components. At this step, we create for each bloc (sub-functional entity) of the high level model a finite state automaton.
The result of the application of the low-level model on the DUP instruction is illustrated in Fig 7. In the low level model a transition between two states of the automaton represents the flow of failure information.

There are different failures possible scenarios which are represented in the generated list. *Ldef* represents a possible failure scenario.

Ldef =
{*TOS.End(true).Bf.Incorrect_shunting.NOS.End(true).*
P_DSP.dM.Stuck-At-Fault_state.Busy_state}
The word 'TOS.End(true)' means that incorrect information from TOS is propagated to the next bloc. The word 'Bf.Incorrect_shunting' represents a fault in the shunting of the MUX caused by a Bit-flip (Bf). Finally, the word 'P_DSP.dM. Stuck-At-Fault_state.Busy_state' means that hardware failure can cause a problem in stack pointer and thereafter a busy state report.

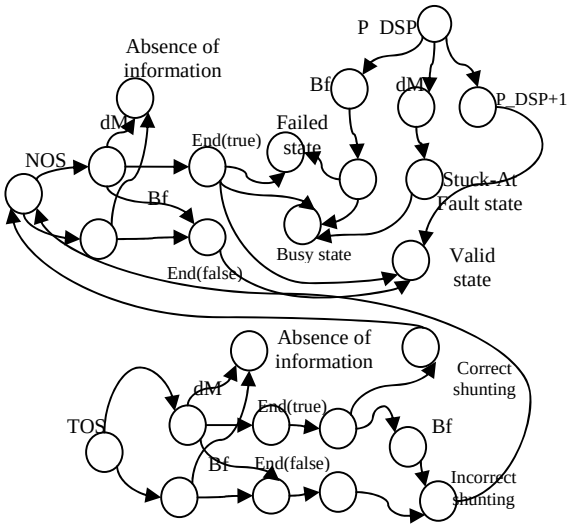

Fig 7. Low-level model for DUP instruction

There are six functional dependability modes, in which we speak about level of credibility of information [13].

- Mode 1: All works well, correct result and no failure detected;
- Mode 2: Looking for Availability, incorrect result, failure detected but tolerated;
- Mode 3: Looking for Reliability, incorrect result, failure detected but not tolerated;
- Mode 4: incorrect result, failure not detected;
- Mode 5: Spurious Shutdown, correct result, failure detected;
- Mode 6: Stop of System, Absence of result.

After generation of all the characteristic lists (scenarios) from low level model, we gather them by family according to correct operation modes (mode 1), failures detected (mode 2 and mode 3), not detected (mode 4), failure tolerated (mode 2) or not tolerated (mode 3), the spurious shutdowns (mode 5), latent errors (mode 4), worst cases (mode 4), as well as the absence of information (mode 6). These scenarios make it possible to calculate the probability of occurrence of these failure modes. In practice, we transform these lists to fault trees. Fault tree analysis is recently a widely accepted technique to assess the probability and frequency of system failure in many industries [14]. The analysis performed on fault tree can be either qualitative or quantitative. Fault tree analysis, based on binary decision diagram (BDD), is a technique that may be used for programmable electronic control system reliability analysis. However, research works show that fault tree generation algorithms are not sufficiently efficient for programmable and complex systems because of some problems, such as variable ordering and combination of large states (combinatory explosion) [15]. The information flow approach is proposed as solution. This qualitative analysis shows, for instance, which event combinations must occur together to cause a system failure. Concerning the quantitative analysis, when we need to calculate the probability of the event, the characteristic lists generated by the information flow approach is transformed to fault trees. The probability of top element event occurring is calculated from the probabilities of the basic events. Tools such as Aralia1 can, in many cases, give results more accurate than conventional tools because it's running 1000 times faster [16].

After the translation of the sub-functional entities of the high level model in the finites states automaton, it is necessary to calculate the transition failure rate to achieve each final state. We use as tools the Markov process. This method can account the common cause failures, multiple failure states, and variable failure rates. The arcs values represent the failure rates of the information flow between two states. These values represent also the matrix elements of this Markov process. The values calculated by the Markov process replace the different probabilities of the basic elements in the fault trees.

The following table gives the results of the DUP instruction probability according to different functional mode:

Table 1. Different probabilities for each functional mode

Probability of existence in:	**Mode 2**	**Mode 3**	**Mode 4**	**Mode 5**
DUP	$72\ 10^{-3}$	$9\ 10^{-5}$	$8.99\ 10^{-10}$	$4.9\ 10^{-22}$

According to this numerical table, we can easily note that the probability of mode 2 (failure detected but tolerated) is higher than the probability of the mode 3 (failure detected but not tolerated). This shows the advantage of the software technique and its efficiency to tolerate failures.

In the next paragraph, this study is generalized on the remainder instructions.

3.3 Evaluation of the sorting program without fault tolerance

The main goal is to know at anytime the values of probabilities of existence on the six functional modes for the whole program. This can be done by exploiting the results from the simple instruction, and the routines execution probabilities.

The probability of the top element in Fig 8 is easily calculated by the assessment of the basic elements probabilities (the probabilities of the sub-trees top-elements). The subdivision of the global fault tree in the sub-fault trees is a practical and effective way to calculate global probability. The calculation of each dysfunction

mode probability of the program requires the replacement of each sub-fault tree element (Fig 9) by its probability for the same dysfunction mode.

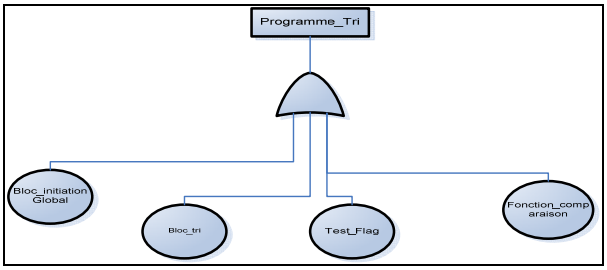

Fig.8. Fault tree of sorting program without tolerance

For example, the probability of basic element 'Flag_Test' is given by the flowing sub-tree as shown in Fig 9.

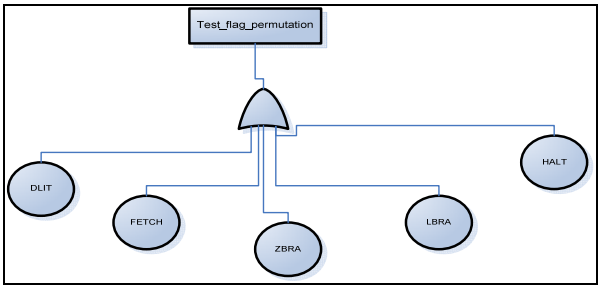

Fig.9. Fault tree of the Test_Flag bloc

In this sub-fault tree, we should find the probability of different assembler instructions which composed this tree with the same manner than DUP instruction detailed previously. The results from this modeling are illustrated by the table 2. In this table, we can assume that the probability to be in mode 3 is important. It means that our architecture is not able de tolerant some fault. By the way the probability in the mode 2 is null because we don't implement yet the recovery technique. In the next step of this work, we show how we can change these results by adding the proposed technique of tolerance.

Table 2. Values probability of sorting program without taking into account recovery functions

	Mode 1	**Mode 2**	**Mode 3**
Initialisation_Bloc	$4.894\ 10^{-8}$	0.0	$3.091\ 10^{-3}$
Sorting_Bloc	$1.27\ 10^{-11}$	0.0	$10.5\ 10^{-3}$
Flag_Test	$9.17\ 10^{-6}$	0.0	$4.59\ 10^{-3}$
Comparison_Function	$58.49\ 10^{-8}$	0.0	$7.9\ 10^{-4}$
Sorting_program	**$7.49\ 10^{-6}$**	0.0	**$6.67\ 10^{-2}$**

4. Sorting program with fault tolerance strategy

We consider the example of the bubble sort program above, the recovery strategy is translated by the addition of the functions of backup and recovery that we model next.

As shown in the organizational chart of the bubble sort program algorithm (fig 10), the integration of the recovery function is done before each sorting iteration in order to keep in memory the correct data before eventual failure.

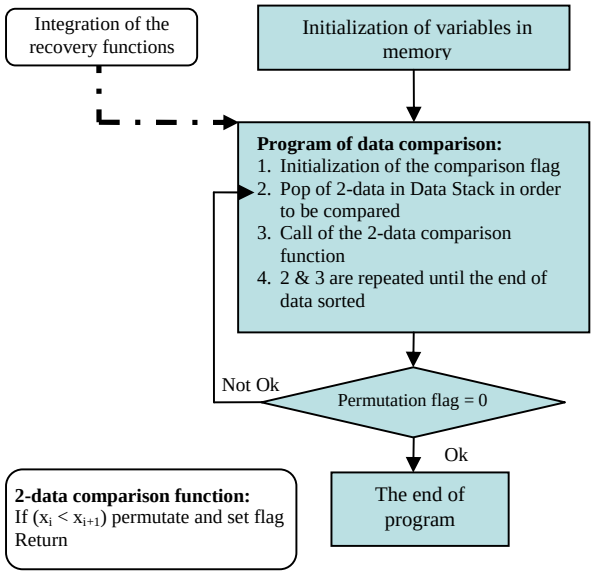

Fig 10. Organizational chart of the bubble sort program algorithm with the taking into account of tolerance

The difference from the previous program is the integration of the recovery functions. This strategy is based on recovering the last dependable state on each error detected and not corrected by hardware. The dependable state is periodically obtained thanks to storage of stack pointers, program counter and top-of-stacks before starting of each comparison cycle. In case of any failure event during the program execution, a restore of saved data is done. After this restoration, the program continues to run with reliable data. The faults tolerance method proposed in this work uses a set of features to save and restore periodically the sensitive elements in stack processor architecture (DSP, RSP, PC, TOS, NOS and TORS). As example, we can illustrate the backup and the restore of the Data Stack Pointer (DSP), given by the following features:

```
---------- Save_DSP ----------
Push_DSP
DLIT @_Sauv_DSP
STORE

---------- Restore_DSP ---------
DLIT @_Sauv_DSP
FETCH
Pop_DSP
```

To evaluate for example the probability of the 'Save_DSP' function, we should have the probabilities of all assembler instructions (Push_DSP, DLIT and STORE).

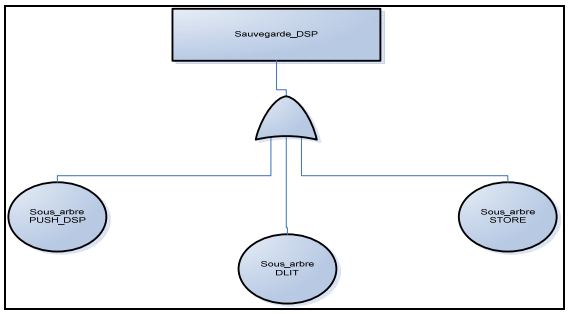

Fig 11. Fault tree of the 'Save_DSP'

The addition of the context storage after each sorting operation requires taking into account the probability values to save and restore correctly at any time the sensitive values (TOS, NOS, TORS, DSP, RSP and PC). The failure probability of the program will be influenced by the failure probability of these functions as showed in the following fault tree in fig 12.

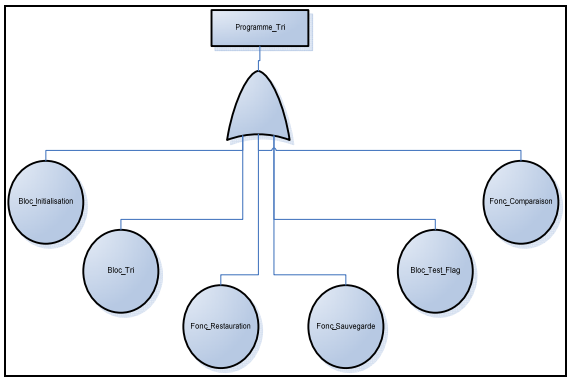

Fig.12. Fault tree of sorting program with tolerance.

The following table summarizes the probability values for each functional mode according to the six mode of operation.

Table 3. Probability values for the sorting program with taking into account recovery functions

	Mode1	**Mode2**	**Mode3**
Initialisation_Bloc	$4.894\ 10^{-8}$	$1.430\ 10^{-2}$	$3.091\ 10^{-3}$
Sorting_Bloc	$1.27\ 10^{-11}$	$1.57\ 10^{-2}$	$10.5\ 10^{-3}$
Flag_Test	$9.17\ 10^{-6}$	$3.55\ 10^{-3}$	$4.59\ 10^{-3}$
Comparison_Function	$58.49\ 10^{-8}$	$3.70\ 10^{-2}$	$7.9\ 10^{-4}$
Save_Function	$7.44\ 10^{-5}$	$1.55\ 10^{-2}$	$2.2\ 10^{-4}$
Restore_Function	$11\ 10^{-5}$	$0.33\ 10^{-2}$	$1.9\ 10^{-5}$
Sorting_Programme	$\mathbf{6.32\ 10^{-5}}$	$\mathbf{1.57\ 10^{-2}}$	$\mathbf{2.39\ 10^{-4}}$

In this table we assume that the probability to be in mode 2 (failure detected but tolerated) becomes important and the probability of mode 3 (failure detected but not tolerated) decreases due to the addition of recovery functions of the sensitive elements in the stack processor architecture. This shows the advantage of the software technique and its efficiency to tolerate failures.

These results can be analysed and compared with results obtained by applying the same approach on the sorting program without tolerance. This comparison is illustrated in table 4.

Table 4. Comparison of the probability values for the sorting program with and without taking into account recovery functions

	Mode 1	**Mode 2**	**Mode 3**
Sorting_Program without tolerance	$7.49\ 10^{-6}$	0.0	$6.67\ 10^{-2}$
Sorting_Program with tolerance	$6.32\ 10^{-5}$	$1.57\ 10^{-2}$	$2.39\ 10^{-4}$

Comparing to the sorting program without tolerance, we notice that for the sorting program with tolerance, the probability to be in mode 1 (correct result and no failure detected) and the probability to be in mode 2 (failure detected but tolerated) have increased. It is a logic result due to the effect of the fault tolerance technique. Whereas, the probability to be in mode 3 (failure detected but not tolerated) has decreased. It means that the non-tolerance of failure is less probable. Thus, it is clearly shown in this table that the implementation of recovery functions has an effect on the final results of probability. The last ones show the advantage of this technique and its efficiency to tolerate failures.

5. Conclusions and perspectives

In this work we have proposed an approach which verifies sufficiently some important reliability parameters of the architecture, complementary with an adequate test-validation strategy, and which achieves the probabilistic reliability target as defined in IEC61508. The interest is to implement, evaluate and enhance fault tolerance mechanisms, and to move our system in the state where tolerance is efficient. Otherwise, the goal is to reduce the probability of mode 3 (failure detected but not tolerated), which is in close relationship with the increase of the information credibility.

We have firstly detailed our global reliability strategy. Because our priority is reliable system design, an implemented software protection technique is evaluated using RTL-VHDL modeling of the instruction set and using the information flow approach. This approach is composed of a high-level model and a low-level model. The first one makes easy the modeling of the architecture hardware resources by the appropriate sub-functional entities. The second one consists of creating a finite state automaton for each high level entity.

An already developed stack processor emulator evaluates the internal states and the running duration and serves for architecture fine-tuning. Surely, benchmarks corresponding to processor embedded programs are developed in order to be tested in the emulator. Because our priority is the dependability and in order to measure its impacts on time performance, we have implemented in benchmark a software protection method: when a perturbation, susceptible to disturb the processor, appears, this one loops in error recovery mode and become no more functional until it returns to a stable and sure state after the occurrence of the error generator event. This technique allows determining the border between the two types of protection (hardware and software).

In order to evaluate this software protection method, we have introduced the information flow approach. We have firstly applied this approach on one assembler instruction to secondly generalize it for all the instruction set. Thus, the probabilities of existence on different functional are calculated for a bubble sort program embedded on the stack processor emulator. Then, the same probabilities are also calculated but with taking into account the fault tolerance strategy. The results are analysed and compared.

The conclusions taken from the numerical results are varied. Before the implementation of the recovery strategy, the blocs program '*initialisation_Bloc*', '*Sorting_Bloc*', '*Test_Flag_Bloc*' and '*Comparison_Function_Bloc*' have a null probability of being in mode 2 (failure detected but tolerated), and an important probability to be in the mode 3 (failure detected but not tolerated). After the implementation of the fault tolerance techniques, we conclude the following items: the probability of mode 2 for these program blocs becomes non-null and more important to the probability of mode 3. It means that the recovery functions effect (backup and restore) is inevitable. The instructions, that compose these features, are the cause of the probability increase of mode 2 of those functions and consequently of the whole program. The functions of recovery (backup and restore) are composed by instructions which are more probable to be in mode 2 than to be in mode 3.

Moreover, the information flow approach has showed its ability to evaluate the hardware-software architecture. Even after the implementation of the tolerance strategy, our approach remains efficient to validate the impact of this policy of tolerance and avoidance of transient faults. Through the collaboration with other research teams in our consortium, we are able to achieve results in the design phase, which is used to prevent certain consequences closely related to the safety and security systems.

6. References

[1] H. Belhadaoui, M. Jallouli, B. Dubois, O. Malassé, K. Hamidi, V. Idasiak, J-B. Kammerer, L.Hébrard, F. Monteiro, C. Diou, M. Hehn, J-F. Aubry, H. Medromi, F. Braun, A. Dandache, S. Piestrak and B. Lepley "Instrumentation sûre de fonctionnement - Une synergie multidisciplinaire", *4ème Colloque Interdisciplinaire en Instrumentation* (C2I 2007), Octobre 2007.

[2] M. Jallouli, C. Diou, F. Monteiro and A. Dandache "Stack processor architecture and development methods suitable for dependable applications", in Proc. *3rd International Workshop on Reconfigurable Communication Centric System-On-Chips* (ReCoSoC'07), June 2007.

[3] K. Hamidi, O.Malassé, J.F. Aubry "Coupling of information-flow aggregation method and dynamical model for a more accurate evaluation of reliability", *European Safety and Reliability Conference*, ESREL, June 2005.

[4] Z.Lei, H.Yinhe, L.Huawei, L.Xiaowei, "Fault Tolerance Mechanism in Chip Many-Core Processors", *Tsinghua Science and Technology*, ISSN 1007-0214 30/49, vol. 12, No. S1, July 2007 pp.169-174.

[5] A.Li, B.Hong, "Software implemented transient fault detection in space computer", *Aerospace Science and Technology 11*, 2007 pp. 245–252.

[6] Kishor S. Trivedi, Jogesh K. Muppala, Steven P. Woolet and Boudewijn R. Haverkort, "Composite performance and dependability analysis", *Performance Evaluation*, vol. 14, Issues 3-4, February 1992 pp. 197-215.

[7] Norme CEI 9126. – Génie du logiciel – Qualité des produits – Partie 1 : Modèle de qualité, Genève 2001.

[8] F. Vallée et D. Vernos, "Le test et la fiabilité du logiciel sont ils antinomiques?" *12ème Colloque national de Fiabilité et Maintenabilité*, Montpellier, 2000.

[9] Jing-An Li, Yue Wu, King Keung, Ke Liu, "Reliability estimation and prediction of multi-state components and coherent systems", *Reliability Engineering and System Safety 88*, 2005 pp. 93-98.

[10] Mile K. Stoj cev, Goran Lj. Djordjevi c, Tatjana R. Stankovij "Implementation of self-checking two-level combinational logic on FPGA and CPLD circuits", *Microelectronics Reliability 44*, 2004 pp. 173–178.

[11] A. Rauzy, "Mode automaton and their compilation into fault trees", *Reliability Engineering and System Safety*, 2002.

[12] C.Bolchini, R.Montandon, F.Salice and D.Sciuto, "Finite State Machine and Data-Path Description", *IEEE Transactions on very large scale integration (VLSI) systems*, vol.8, No.1, February 2000 pp. 98-103.

[13] IEC 61508, 1999 Functional Safety of Electrical/ Electronic/Programmable Electronic Safety-Related Systems, Part 1-7. International Electrotechnical Committee.

[14] M.Yuchang, L.Hongwei, Y.Xiaozong, "Efficient Fault Tree Analysis of Complex Fault Tolerant Multiple-Phased Systems", Tsinghua Science and Technology, ISSN 1007-0214 22/49 vol. 12, No. S1, July 2007 pp.122-127

[15] J D.Esary, H.Ziehms, "Reliability analysis of phased missions" In *Proceedings of Reliability and Fault Tree Analysis*, Philadelphia, USA, 1975 pp. 213-236.

[16] Group Aralia, "Computation of prime implicants of a fault tree within Aralia" In *Proceedings of the European Safety and Reliability Association Conference*, ESREL'95, Bournemouth, UK, 1995: 190-202.