POLITECNICO DI TORINO
Repository ISTITUZIONALE

Automated Synthesis of SEU Tolerant Architectures from OO Descriptions

Original
Automated Synthesis of SEU Tolerant Architectures from OO Descriptions / Chiusano, S.A., DI CARLO, S., Prinetto, P.E.. - STAMPA. - (2002), pp. 26-31. (IEEE 8th International On-Line Testing Workshop (IOLTW) Isle of Bendor, FR 8-10 July 2002) [10.1109/OLT.2002.1030179].

Availability:
This version is available at: 11583/1408856 since:

Publisher:
IEEE Computer Society

Published
DOI:10.1109/OLT.2002.1030179

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

24 June 2026

Automated Synthesis of SEU Tolerant Architectures from OO Descriptions

S. CHIUSANO, S. DI CARLO, P. PRINETTO
POLITECNICO DI TORINO
Dipartimento di Automatica e Informatica Torino, Italy
E-MAIL: {chiusano, dicarlo, prinetto}@polito.it

Abstract

SEU faults are a well-known problem in aerospace environment but recently their relevance grew up also at ground level in commodity applications coupled, in this frame, with strong economic constraints in terms of costs reduction. On the other hand, latest hardware description languages and synthesis tools allow reducing the boundary between software and hardware domains making the high-level descriptions of hardware components very similar to software programs. Moving from these considerations, the present paper analyses the possibility of reusing Software Implemented Hardware Fault Tolerance (SIHFT) techniques, typically exploited in micro-processor based systems, to design SEU tolerant architectures. The main characteristics of SIHFT techniques have been examined as well as how they have to be modified to be compatible with the synthesis flow. A complete environment is provided to automate the design instrumentation using the proposed techniques, and to perform fault injection experiments both at behavioural and gate level. Preliminary results presented in this paper show the effectiveness of the approach in terms of reliability improvement and reduced design effort.

1. Introduction

Spacecraft and spacecraft designers are being pushed to use enabling or emerging commercial technology to meet high science data performance in increasingly smaller and lower cost spacecraft [1]. The benefits may include: higher gates density, increased speed and performance, easier system development process using COTS development and test equipment, and decreased lead times versus rad-hard (RH) approaches.

The design of SEU tolerant architectures is a well-known problem in aerospace environment, where radiations, such as alpha particles and cosmic rays, can cause transient faults in electronic systems. *Single-Event Upset* (SEUs) faults mainly affect flip-flops, memory cells, registers and latches causing an undesired change of state (*bit-flip*) in the storage elements [2]. The SEU effects have been well investigated in literature and can be classified in: (i) *fail silent* condition, whether the fault is masked; (ii) *fail silent violation*, when the system outputs incorrect values; (iii) *system crash* when the system stop working.

Unfortunately IC manufacturers are being driven by a market of which the space community is a very small portion. Because of this, commercial approaches must be evaluated and sometime modified to meet performance and reliability requirements of spacecraft applications [1]. In addition, with the increasing use of nanometre technologies, due to electromagnetic interference and power supply glitches, SEU effects become a relevant problem also at ground level. In this scenario SEU tolerant architectures design becomes mandatory also in commodity applications coupled, in this frame, with strong economic constraints in terms of costs reduction [3].

Many hardware techniques have been proposed to develop SEU tolerant architectures. They include *Signature monitoring* for low cost concurrent error detection for FSMs; *On-line monitoring* of reliability relevant parameters; *Self-Checking*; *ON-line BIST techniques*; *Scan Paths* exploitation [4].

With the introduction of hardware description languages as VHDL and the possibility of describing systems at RT-level several researches have been performed to introduce SEU tolerant structures directly at this description level. The reason relies on the fact that the VHDL code can be easily analysed and modified before synthesis introducing safe operators and functional blocks. Acting in the earlier steps of the design flow the designer can trade-off costs and dependability improvements, globally reducing the developing time. Possible approaches include synthesis techniques for self-checking combinational circuits [4] [5][6], automatic generation of self-checking data paths using CAD tools [7]; libraries of VHDL components designed for high circuit reliability and availability [8] [9].

Concurrently, the massive use of micro-processor based systems lead to the development of alternative solutions named *Software Implemented Hardware Fault Tolerance* (SIHFT) techniques [10][11][12]. These software techniques are able to protect the memory space where program data and instructions are stored. The basic idea is to apply a set of transformation rules to the original code to obtain a new version, functionally equivalent but SEU tolerant. Mainly they include *Error Detection by Duplicated Data* (EDDD) approaches

Proceedings of the Eighth IEEE International On-Line Testing Workshop (IOLTW'02)
0-7695-1641-6/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

[13][14], executing operation on multiple copies of the same data and comparing the produced outputs to detect the fault occurrence, and *Control Flow Checking by Software Signatures* (CFCSS), based on signatures assigned to each code block a-priori and re-computed run-time to verify the flow execution [15][16][17].

The actual trend in the design flow is to move up into the hierarchy, describing the circuit at the behavioural level. These high-level descriptions are synthesizable descriptions but at the same time they are very close to design specifications. In this context the boundary between software domain and hardware domain looses weight and the reuse of SIHFT techniques in the hardware components descriptions becomes possible.

The present paper works in this area and proposes an automate approach to reuse SIHFT techniques in high level hardware descriptions. The goal is to show the effectiveness of the approach in terms of reliability improvement and reduced design effort. The target environment for behavioural level designing is the *SystemC* environment.

SystemC is a modeling platform consisting of C++ class libraries and a simulation kernel to design at the system-behavioral and register-transfer-levels. SystemC represents a de facto standard for system-level design and is supported by a collaborative effort among a broad range of companies named *Open SystemC Initiative* (OSCI). In comparison with traditional hardware description languages, C++ appear to be the most suitable approach for behavioural-level design. It provides the control and data abstractions necessary to develop compact and efficient descriptions using the power of an *Object Oriented (OO)* language. In addition, most systems include both hardware and software units, the last ones typically described in C/C++. Finally, most designers are familiar with this language and a large number of development tools is available coming from the software area.

The proposed approach includes a SystemC-based class library, named *SAFE Library*. It *extends* the original SystemC class collection with constructs characterized by high dependability properties. The original design description is instrumented with the SAFE Library before the synthesis flow in order to obtain a *SAFE Description* compatible with all the synthesis tools supporting SystemC. In the paper we propose one implementation of SAFE library, but the approach can support any SHIFT technique.

A *SystemC Environment for Applications* (SEA) tool is provided to *automate* the insertion of the SAFE Library. The tool also allows validating the modified design description and performing fault injection experiments both at behavioural and gate level. Using the SEA tool the designer can efficiently trade-off costs and dependability properties tuning the optimal SEU tolerant solution for the target design.

This paper is organized as follows: Section 2 analyzes the reuse of SIHFT techniques in the hardware domain whereas Section 3 presents the proposed design flow and the SEA tool. Section 4 reports some experimental results to validate the proposed approach and Section 5 draws some conclusions.

2. Using SIHFT techniques in HW designing

With the introduction of *Object Oriented* (OO) design languages like C++, the high-level description of hardware components becomes very close to a standard software program. Well-known and verified algorithms written using C/C++ can be synthesized and implemented as ASIC cores or mapped into FPGA components with a very low design effort.

However, software programs and high-level hardware descriptions have different final implementations, and thus so far different approaches have been proposed to achieve SEU tolerance in the hardware and software domains.

The OO code of a software program is translated by a compiler in a sequence of instructions executable in a given micro-processor platform. A memory accessible by the micro-processor stores instructions and data needed during the computation process. Thus, to deal with SEU occurrence in the memory, software techniques (SIHFT) target two aspects: the generation of *incorrect data* (1) and *illegal executions of the program flow* (2). The former are due to the occurrence of SEUs in memory locations storing data whereas the latter are usually caused by SEUs which corrupt *jump* or in general *control flow* instructions in memory locations storing code. Typically SIHFT techniques address (1) via *data protection* and (2) via *control flow monitoring*. The basic idea is to modify the original code introducing redundancy. The proposed approaches differ in terms of achieved dependability levels and introduced performance degradation (delays and area).

The OO code describing hardware components is converted through the synthesis process into a networks of combinational blocks and memory elements, executing the expected computation process.

To reuse the SIHFT techniques in the hardware domain some preliminary analysis and considerations should be made. The basic idea is to define a link between the final hardware implementation and the high-level design description. First, the target dependability properties have to be identified at the hardware level. Then, they have to be mapped into the high-level design description. Finally, SIHFT techniques have to be adapted to obtain code synthesizable and able to achieve the target dependability properties in the hardware implementation.

IEEE COMPUTER SOCIETY

To achieve SEU tolerance, the memory elements available in the post-synthesis description have to be protected. They include memory elements (flip-flops) instanced to store data and memory elements implementing the control logic. SIHFT techniques developed to target (1) can be efficiently exploited to achieve data protection in the hardware domain, but solutions for (2) are not directly applicable.

SIHFT approaches for *data protection* typically act on program variables inserting redundant information such as error code and data replication, and checking all the assignments, controls and evaluations of variables to keep status up. In the high-level description of hardware components, *variables* store data and are mapped in memory elements during the synthesis flow. Therefore, SIHFT techniques for (1) can be easily reused to achieve data protection in the hardware domain.

Several solutions have been proposed in software for *control flow monitoring* (2). The basic idea is to split the original program into elementary blocks and for each block off-line compute a reference *signature* representing the correct execution of the block. At run time the signatures are calculated again and compared with the golden ones. Since the signatures are computed on portions of program machine code, these approaches are not directly applicable to hardware descriptions where the concepts of instruction, addresses and opcode have no meaning in the final implementation.

To reuse control flow monitoring techniques in the hardware domain an analogy with the software domain should be found. The synthesis process inserts some logic (both combinational parts and memory elements) to implement the control. This logic is not explicitly defined in the high level description, but it is inferred by the synthesis process based on the computation flow specified in the high-level description, and some additional synthesis constraints (e.g., available units and synthesis for minimum area or delay). The control logic is usually implemented as a Finite State Machine (FSM) where the memory elements influence the states evolution. The sequence of states of the FSM has a strong analogy with the control flow of a software program. Therefore, SIHFT techniques for (2) can be an effective solution to protect control logic in the hardware design implementation. They only have to be modified in order to work without resorting to the concept of instruction opcode or memory addresses.

Starting from this analysis, we propose an approach which works in two different directions:

- ***Make safe variables***, to detect and /or correct the SEU occurrence on them. The approach generates redundant copy of variables or of part of them and check the consistence of the copy.
- ***Make safe control on flow***, proposing an approach independent from program machine code. It is implemented as a module call *Agent* able to check the correct evolution of the FSM implementing the control logic.

Despite the approach used to protect variables is intuitive some more words must be spent concerning control flow protection. The *Agent* is a relatively simple unit running concurrently with the given design. The Agent is able to receive information about the current state of the control logic and check the correct evolution of them. It can be easily implemented using OO languages. The original code has to be slightly modified inserting so called *checkpoints* in correspondence of branch instructions in order to transmit status information to the Agent. This implementation allows reusing most of the concepts and theory defined and well proved in the software control flow checking field.

3. The proposed design flow

To automate the insertion of dependable structures and validate the concepts sketched in Section 2 a complete design environment has been set up.

As target platform we selected the SystemC v. 1.2 environment, the most widely used in high-level designing. SystemC relies on a library of classes and basic blocks to describe hardware systems at the high-level. In our approach we extend the SystemC Library defining a *SAFE Library* which includes constructs to achieve data protection and control flow monitoring. The implementation takes advantage of all the facilities offered by C++ language like template, classes inheritance, and operators override [18].

Figure 1 shows the proposed design flow. It is structured in two design steps, named *High-level Design Instrumentation* and *Synthesis Process*, and two validation steps based on fault injection experiments, at the high and gate-level.

First, the given high-level design description is properly instrumented in order to obtain its reliable version (*High-level Design Instrumentation*). The modification process is based on the content of the *SAFE Library* and is driven by the designer who selects the portions of the code to modify. Moreover, the designer can trade-off dependability improvements and implementation costs choosing, out of the SAFE Library content, the constructs to apply. As an example, most critical variables can be modified inserting constructs to achieve fault correction; the simple fault detection can be implemented for the remaining variables, or even a certain subset of variables can be untouched.

To avoid long backtracks in the design flow, the developed environment allows validating at early design steps the dependability properties of the instrumented design. Fault injection experiments performed on the safe version of the high-level design description (*High-level Fault Injection*) check whether the performed modifications allow meeting the target reliability

IEEE COMPUTER SOCIETY

constraints. If not, the designer can restart the process, modify differently the original design description, and tune the optimal design solution. Otherwise, the design is synthesized and a gate-level SAFE implementation is gathered (*Synthesis Process*). Fault injection experiments on the gate-level description (*Gate-level Fault Injection*) check that constructs inserted at the high-level maintain their functionality through the synthesis flow. The aim is to verify that the dependability properties of the SAFE gate-level description are the expected ones and correspond to the high-level SEU tolerance previously computed.

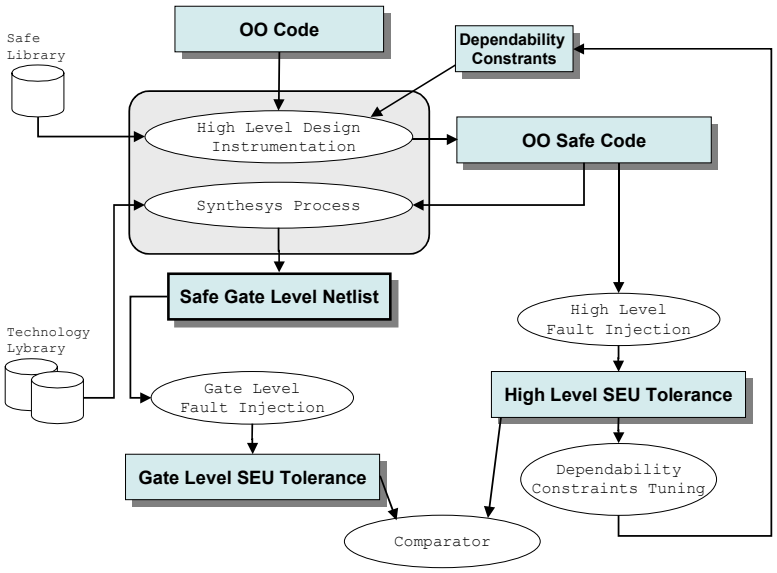

Figure 1: *The proposed design flow*

4. Experimental results

Performed experiments show that the reuse of SHIFT techniques in the hardware domain is a feasible approach.

A tool named *SystemC Environment for Applications* (SEA) has been implemented to perform the design instrumentation and the fault injection experiments.

The SystemC design descriptions have been synthesized using CoCentric™ SystemC Compiler v. 2000.11. All the optimisation options have been disabled during the synthesis process, to preserve the modifications introduced in the safe version of the design description. An in house-developed technology library for synthesis has been used, including elementary gates and D flip-flops.

The used CoCentric™ SystemC Compiler release do not yet completely support all the SystemC constructs. For this reason, the SEA tool implements a *refinement* step which automatically elaborates the code and generate a fully synthesizable version.

Table 1 summarises the characteristics of the considered benchmarks. Since in the present paper the aim is the validation of the approach, we considered relatively simple examples as test cases. They are four data-dominated designs, named *Alarm*, *Front_cnt*, *Counter*, and *Singen*, in which most of the available logic is devoted to data storing and data manipulation, and three control-flow dominated designs named *Bouble Sort*, *2D Gaussian Distribution*, and *FFT*, in which the most critical SEUs are the ones affecting the control flow execution.

The experiments aim at verifying the correspondence between design SEU tolerance measured at the high and gate-level, and evaluating the impacts of the introduced modifications in terms of performance degradation.

Table 2 and Table 3 report the high-level and gate-level dependability measures when the benchmarks are instrumented with the SAFE Library. About 10,000 fault injection experiments were run, and the design *SEU tolerance* is computed as:

$$\frac{\#\cdot\text{of}\cdot\text{detected and corrected}\cdot\text{SEUs}}{\#\,\text{of}\cdot\text{injected}\cdot\text{SEUs}}\%$$

In Table 2, for each benchmark all the available variables have been protected inserting the SEU detection capability, in the first two columns, and the SEU correction in the remaining two.

IEEE COMPUTER SOCIETY

Data-dominated benchmarks			Control-flow dominated benchmarks	
Circuit	**Description**	**Type of Variables**	**Circuit**	**Description**
ALARM	The output signal of the unit blinks for NMAX clock cycles after the start signal is triggered on.	INTEGER BOOLEAN	BOUBLE SORT	Sort an array of elements
FRONT_CNT	This unit verifies at each clock cycle if its input is equal to zero or one, and outputs the number of zeros and one counted.	INTEGER	2D GAUSSIAN DISTRIBUTION	Estimate a 2D Gaussian distribution
COUNTER	It is a |4096| counter with parallel data load input.	INTEGER	FFT	Compute fast Fourier Transformate
SINGEN	It outputs a sinusoidal waveform.	INTEGER A SET OF PRE-COMPUTED VALUES		

Table 1: *Benchmarks*

The introduced modifications guarantee complete fault tolerance at the software level, since SEU occurred in variables are always either detected or corrected. The results gathered at the gate-level still confirm the effectiveness of the approach. The apparent reduction of the SEU tolerant values is mainly due the difference between the high-level and gate-level fault injector. In fact, at high-level fault can be injected in variable storing data, only. On the other hand, at gate-level faults are injected in any point of the circuit, since data and control logic can not be distinguished. Since benchmarks have been instrumented to achieve data protection, SEUs affecting the control logic are neither detected nor corrected, and consequently the fault tolerance values decrease. Being the considered examples data-dominated, the control logic is rather small and therefore the SEU tolerance values at the high-level and gate-level are quite close. Experiments therefore show that SIHFT techniques for data protection can be effectively used in hardware domain.

Table 2 shows that at the gate-level the fault tolerance is higher when resorting to the correction strategy. This increase is due to the synthesis process. The hardware overhead due to the correction strategy is higher then the one due to the detection approach; in addition not all the instanced logic is always in use, but typically some portions are idle in certain time slots. The synthesis process reuses this logic to implement part of the control logic, and as a consequence the data protection is partially extended to the control logic.

Table 3 summarizes the experiments when the *Agent* approach is applied to the three control-flow dominated examples. An ad-hoc mechanism has been implemented to emulate control flow fault injection at high level.

In Table 3, SEU tolerance values are lower then the target 100% due to the *aliasing* phenomenon. The aliasing is related to the block size and the signature computation function, and cause masking the SEU occurrence. Mainly, the execution of a certain block can generate a correct signature even when the flow execution was not correct.

The SEU tolerance values further decrease at the gate-level for the noise introduced by the fault injection experiments. Being the benchmarks control-flow dominated the two results are anyhow quite close.

Example	SEU detection mode		SEU correction mode	
	High-level descr.	**Gate-level descr.**	**High-level descr.**	**Gate-level descr.**
ALARM	100%	91%	100%	97.2%
FRONT_CNT		88.6%		89.7%
COUNTER		86.9%		89%
SINGEN		88.8%		90%

Table 2: *SEU tolerance via data protection*

The results in Table 3 show that the Agent can efficiently monitor the control flow. At the same time they point out that more investigations are needed for an optimal use of the Agent, in order to avoid SEU masking.

Finally, experiments have been performed instrumenting all the examples with both data protection and control flow monitoring constructs. The obtained SEU tolerance is about the 100% also at the gate-level. The reason of the not full tolerance is mainly a sub-optimal use of the Agent and is not in the lack of data protection.

Concerning the implementation costs, the area overhead is comparable with the one of already proposed hardware SEU tolerant architectures. For the examples in Table 2, where most of the logic is devoted to data, the area of the SEU tolerant design is about two times the original area when the SEU detection capability is applied to all the variables, three times in the case of the SEU correction approach.

Finally, experiments pointed out that good dependability properties can be achieved with a lower cost carefully selecting the variables to protect. In the ALARM example we identified a subset of critical

IEEE COMPUTER SOCIETY

variables more often crossed to propagate values through the design. 92% SEU tolerance is achieved protecting the critical variables, about 34% of the available variables. Acting on the remaining 66% of variables, a similar SEU tolerance is obtained but the introduced area is significantly higher since in this case the number of protected variables is almost doubled.

Example	SEU detection in high-level descr.	SEU detection in gate-level descr.
BUBBLE SORT	73%	69.8%
2D GAUSSIAN DISTRIBUTION	83%	81.4%
FFT	54%	51.3%

Table 3: *SEU tolerance via control flow monitoring*

5. Conclusions and on going work

The present paper analyzed the reuse of Software Implemented Hardware Fault Tolerance (SIHFT) techniques to design SEU tolerant architectures. Two aspects have been targeted: data protection and control flow monitoring. Well-known SIHFT techniques have been customized to make them synthesizable and to finally generate a gate-level description with improved dependability properties. A complete development environment has been set up to automate the design instrumentation using SIHFT techniques and evaluate the results via high-level and gate-level fault injections. Preliminary results presented in this paper showed the effectiveness of approach. Currently we are investigating alternative SIHFT techniques, to evaluate the trade-off in terms of dependability improvements and performance degradation. Moreover, metrics are under study to identify the most critical areas in the given system, and thus optimize the design modifications.

6. Acknowledgments

The authors wish to thank Synopsys Inc. which kindly provides us the CoCentric SystemC Compiler to run the experiments. Moreover, the authors thank C. Gay and L. Bianchi for the fruitful discussions and for implementing the SEA tool.

7. References

[1] K.A. LaBel, M. M. Gates, A. K. Moran, P.W. Marshall, J. Barth, E. G. Stassinopoulos, C. M. Seidleck, C. J. Dale, *Commercial Microelectronics Technologies for Applications in the Satellite Radiation Environment*, IEEE Aereospace Application Conference, 1996.

[2] F. W. Sexton, *Measurement of Single Event Phenomena in Devices and ICs*, IEEE NSREC Short Course, pp. III-1 III-5, 1992.

[3] M. Nicolaidis, Y. Zorian, *On Line Testing for VLSI – A Compendium of Approaches*, Journal of Electronic Testing, Theory and Application (JETTA), Vol. 2, Nos. 1/2, Feb-Apr. 1998, pp. 7-8.

[4] K. De, C. Natarajan, D. Nair, P. Banerjee, *RSYN: a system for automated synthesis of reliable multilevel circuits*, IEEE Transaction on VLSI Systems, pp. 186-195, June 1994

[5] N. K. Jha, S. J. Wang, *Design and Synthesis of Self-Checking VLSI Circuits*, IEEE Transaction on CAD, vol. 12, No. 6, pp. 878-887, June 1993

[6] F. Salice, M. Sami, D. Sciuto, *Synthesis of Multilevel Self-Checking Logic*, IEEE Int. Workshop on defect and fault Tolerance in VLSI, October 1994

[7] B. Hamadi, H. Bederr, M. Nicolaidis, *A tool for automatic generation of self-checking data paths*, IEEE VLSI Test Symmposium, pp. 460-466, April 1995

[8] Vargas, F.; Amory, A., *Recent improvements on the specification of transient-fault tolerant VHDL descriptions: a case-study for area overhead analysis*, 13th Symposium on Integrated Circuits and Systems Design, 2000, pp. 249 -254

[9] Stroud, C.; Ding, M.; Seshadri, S.; Kim, I.; Roy, S.; Wu, S.; Karri, R., *A parameterized VHDL library for on-line testing*, IEEE International Test Conference, 1997, pp. 479 –488

[10] P. P. Shirvani, N. R. Saxena, E. J. McCluskey, *Software-implemented EDAC protection against SEUs*, IEEE Transaction on Reliability, Vol. 49, Issue 3, pp. 273-284, Sept. 2000.

[11] P. P. Shirvani, N. Oh, E. J. McCluskey, *Software-Implemented Hardware Fault Tolerance Experiments COTS in Space*, Center for Reliable Computing, Stanford University, Dec. 2000.

[12] V. Strumpen, *Portable and Fault-Tolerant Software Systems*, IEEE Micro, pp. 22-32, September-October 1998.

[13] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, M. Violante, *Soft-error Detection through Software Fault-Tolerance techniques*, DFT'99: IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, November 1-3, 1999 - Albuquerque, New Mexico, USA, pp. 210-218

[14] A. Benso, S. Chiusano, P. Prinetto, L. Tagliaferri, *A C/C++ source-to-source compiler for dependable applications,* IEEE Dependable Systems and Networks, 2000, pp. 71 -78

[15] G. Miremadi, J. Karlsson, U. Gunneflo, J. Torin, *"Two software techniques for on-line error detection"*, 22th International Symposium on Fault-Tolerant Computing (FTCS-22), pp. 328-335, July, 1992.

[16] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, J.A. Abraham, *"Design and Evaluation of System-Level Checks for on-line Control Flow Error Detection"*, IEEE Transaction on Parallel and Distributed Systems, Vol. 10, No. 6, pp. 627-641, June 1999.

[17] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, L. Tagliaferri, *Control Flow Checking via Regular Expressions,* to be presented at Asian Test Sympsyum, Nov. 2001.

[18] B. Stroustrup *The C++ Programming Language,* II edition, Addison-Wesley, 1991.

IEEE COMPUTER SOCIETY