A Low-Redundancy Approach to Semi-Concurrent Error Detection in Data Paths

Anna Antola, Vincenzo Piuri, Mariagiovanna Sami
Department of Electronics and Information, Politecnico di Milano
Piazza Leonardo da Vinci 32, 20133 Milano, Italy
Phone: +39-2-2399-3516, Fax: +39-2-2399-3411, Email: {antola, piuri, sami}@elet.polimi.it

Abstract

A high-level synthesis approach is proposed for design of semi-concurrently self-checking devices; attention is focussed on data path design. After identifying the reference architecture against which cost and performances should be evaluated, a simultaneous scheduling-and-allocation algorithm is presented, allowing resource sharing between nominal and checking data paths. The algorithm grants that the required checking periodicity is satisfied while minimizing additional costs in terms of functional units. Risk of error aliasing due to resource sharing is analysed.

1: Introduction

The problem of defining High-Level Synthesis techniques for design of self-checking or even self-diagnosing ASICs has been recently approached from some different points of view; attention has been focused in all instances on scheduling, allocation and binding of Data Paths, it being considered that FSM-related techniques could be separately adopted for the control unit (e.g., see [1]). Thus, in [2] and in [3] the problem of autonomous error detection and recovery from *transient* faults was in particular taken into account; other authors dealt on the contrary with *permanent* faults, the principal aim being that of achieving detection (and, possibly, location) of faults with impact on performances as low as possible (ideally, nil) and with limited area redundancy.

Solutions presented in the literature have taken into account *concurrent* error detection, such that checking is performed - at the latest - at the end of the complete execution of the DFG. In [3,4], the authors presented a code-based solution valid for DFG’s consisting of *arithmetic operations* only; scheduling and allocation algorithms were proposed allowing to grant detection of any single fault (within a fairly comprehensive fault model) with low error latency and limited area redundancy, checking being performed at suitable intermediate points (as well as on primary outputs) so as to minimise the number of checkers while avoiding aliasing. In [4] a technique was introduced granting not only fault detection but also *fault location*, that exploits at each control step the presence of unused non-redundant resources to repeat operations in the DFG; in this case, the null-redundancy requirement (at least, as far as functional units are concerned) leads to the drawback that for some operation neither checking nor location are feasible.

In the present paper, we introduce a *semi-concurrent* self-checking technique. The rationale justifying adoption of semi-concurrent testing derives from the low fault occurrence rates granted by present silicon technologies, that - excepting the case of extremely severe operation environments - makes it reasonable to perform checking operations not concurrently with each nominal operation cycle but periodically, in correspondence of each *N-th* operation cycle (*N* being suitably defined with respect to the application and the expected fault occurrence rate). Such techniques have been proposed in the past for regular architectures - typically, linear arrays or rectangular arrays [5], [6] - where they allowed to exploit a limited number of redundant units to cyclically duplicate operation of nominal units so as to achieve detection as well as identification of faults; *nominal* input data are always used for checking purposes, so that error latency actually derives not only from the periodicity of checking operations but also from possible non-activation of existing faults by the present input data. The results achieved by these approaches thus offer a compromise between costs (expressed in terms of area redundancy) and checking efficiency (expressed mainly in terms of error latency). Obviously, semi-concurrent techniques are effective for systems that operate on a continuous flow of data; such systems cannot be excluded from nominal operation in order to undergo an off-line test phase (even if by means of BIST techniques, see [7]), and on the other hand the flow of input data can be considered as a long sequence of random test vectors, leading to acceptable fault coverage.

The semi-concurrent self-checking approach proposed in the present paper relates to arbitrary application-specific systems, described by a Data Flow Graph (*DFG*) on which no constraints are given either in terms of structure or in terms of allowable operation types. We concentrate - as authors of previously recalled papers have done - on synthesis of the Data Path; while an implicit form of checking is performed on the controlling FSM as well, through the commands received by the Data Path and the results produced, if such checking is not deemed adequate a totally concurrent self-checking FSM can be designed by proper rules [1]. The aim of our approach can be summarised as follows:

> for a given optimum nominal scheduling and allocation of a DFG, and for a given *periodicity of checking*, synthesise a modified Data Path that will allow semi-concurrent self-checking with minimum redundancy (in terms of functional units) and minimum risk of aliasing

where *periodicity of checking* is defined as the ratio between the number of control steps between two subsequent self-checking iterations and the number of control steps required by the nominal schedule, while *aliasing* occurs whenever the system fails to detect an error present in its results.

In Section 2, we define the fault model and the rules by which periodicity of checking is computed. In correspondence, we present the reference architecture against which results produced by our synthesis technique will be evaluated. In Section 3, we define the synthesis approach for minimum-cost semi-concurrent self-checking data paths; high-level synthesis is performed by means of simultaneous scheduling and allocation. DFGs discussed in current literature on fault-tolerance related topics will be used as running examples. In Section 4, the problem of aliasing will be examined and the guidelines by which the application designer can evaluate the risk of aliasing for the specific application will be introduced.

2: Fault assumptions and reference architecture

The approach here chosen to achieve semi-concurrent self-checking operates by modifying standard high-level synthesis techniques, so that the synthesised data path grants self-checking properties; as a consequence, the fault model adopted is of necessity a *functional* one, technology-independent.

We assume initially that:

a) faults are concentrated in *functional units*; the connection network is fault-free, and so are the registers. Such restriction is consistent with assumptions made by other authors (see, e.g., [8]) and is justified by the relative complexity of functional units with respect to switches and registers; in the final section, anyway, it will be seen that fault assumptions can be relaxed, leading to a more comprehensive fault model;
b) at most a *single fault* (i.e., a single faulty unit) is present in the system. Since the approach relates to run-time checking rather than to end-of-production testing, it is sufficient to choose an adequate frequency for semi-concurrent self-checking operations in order to make the assumption acceptable; in the case of a very complex system, it is also possible to partition it into sub-systems each individually provided with self-checking capacities, thus allowing for presence of a larger number of faults.

Given these assumptions, we define a *reference architecture* against which architectures provided by our high-level synthesis approach will be evaluated. The *nominal architecture* is defined as that initially designed *without* any self-checking capacity, scheduling, allocation and binding being performed on the basis of cost and performance requirements only. We assume here latency as the primary figure of merit, so that the number k_N of control steps required coincides with the length of the critical path, but this does not constitute a restriction for our approach. If by τ we denote length of the clock cycle, latency of the nominal architecture will be $l_N=k_N\tau$. Based on fault information derived from technology as well as from data on operation environment, the *checking periodicity* is evaluated as the minimum time t_C to be allowed between two successive checking actions; since *nominal data* are used in semi-concurrent self-checking, allowance must be made for the possibility that nominal input data will not excite a fault, thus making checking periodicity tighter. Actually, we will refer to the *relative checking periodicity*, computed as the ratio $P = \left\lfloor \frac{t_C}{l_N} \right\rfloor$; if $P=0$, the system is partitioned into subsystems each with latency suitably lower than l_N, and the approach applied to each subsystem individually.

An independent checking architecture is now designed starting from the DFG of the nominal circuit and implementing the lowest-cost data path compatible with P. If initial fault information allows it (i.e., if $t_C >> l_N$), a straightforward resource-constrained solution can be adopted leading to the design with minimum number of functional units; otherwise, a time-constrained solution with t_C as the allowable latency and cost as the secondary figure of merit will be adopted. Let $l_R = k_R\tau$ be the latency of the independent checking circuit; we must add to it the time $l_C = k_C\tau$ required for checking results on the critical paths. Assuming that a checker operates in one control step, k_C depends only on the chosen schedule and on the number of checkers available. Periodicity of checking is satisfied if $t_C \geq l_R + l_C$ (or $P \geq \left\lceil \frac{l_R + l_C}{l_N} \right\rceil$).

Operation of the complete self-checking reference architecture is as follows:

1. The same set of nominal data (*checked* data) is fed to both nominal and checking circuits. The two circuits then operate separately; in particular, after k_N control steps the nominal circuit will receive a new set of input data and start operating on them. The primary outputs produced by the nominal circuit operating on the checked data are stored in a suitable register until the corresponding value has been computed by the checking circuit; then checking is performed and the register in the nominal circuit is freed. *Self-checking checkers* are used to perform checking; the number of checkers depends on the DFG as well as on latency and cost requirements;
2. After k_R+k_C control steps, either a checking operation has detected an error and an error signal is activated, or both circuits are declared fault-free; after P iterations by the nominal circuit, both circuits receive a new set of checked data and step 1. is repeated.

The two circuits do not share any resource, so that in the single-fault assumption if results produced are identical we can safely state that operation of both is error-free (either no fault is present or a possible fault is not excited by the nominal set of input data). In other words, no *aliasing* (by which error-affected results would be considered correct) is possible. Periodicity of checking is satisfied by construction; the total number of resources required - in terms of functional units - is given by the sum of resources in the nominal and in the checking circuit. If a minimum-cost checking circuit has been designed, in particular, we need only add one to each type of functional unit required by the nominal circuit. A rough evaluation of the controlling FMSs' complexity identifies a "checking" FMS with k_R+k_C states and a simple controller granting that both data paths are fed the same data at each Pk_N control steps.

In general, in the nominal circuit not all functional units present will be actually used *in each control step*; based on this consideration, we explore the possibility of reusing the "nominal" functional units for checking operations, so as to lower the cost of the self-checking data path. Constraints ruling such reuse will be described in the next section, where the scheduling-and-allocation algorithm that allows creation of a checking circuit sharing the nominal circuit's resources will be presented.

3: Resource sharing for minimum-cost semi-concurrent self checking

To reduce cost of the semi-concurrent self-checking data path, we envision the possibility of scheduling and allocating both nominal data path and checking data path onto the same, *shared*, resources. More precisely, we attempt to execute two separate DFG's, corresponding to nominal and checking processes, by making use of the nominal data path, increasing its set of registers and modifying its interconnection structure but without increasing its set of functional units or by adding the minimum number of such units. To achieve this goal, a set of necessary conditions must be verified first:

a) to grant that probability of aliasing is much lower than 1, we must ensure that *no operation o_i in the unscheduled DFG will be performed in both nominal and checking data path by the same functional unit f_j*. This implies restrictions on scheduling and allocation of the checking DFG;
b) scheduling of the checking DFG onto the shared resources must be possible within the periodicity P evaluated for the envisioned application.

A preliminary analysis of the nominal data path leads to possible increase of the set of functional units. In particular:

1) whenever, for a given type t_k of functional unit, one instance only is present in the nominal data path, a second instance must be added to achieve condition a);
2) whenever, for a given type t_k of functional unit, all instances are used in each control step of the nominal schedule, a further instance must be added to achieve condition b) above.

Condition 1) grants that proper scheduling and allocation of the checking data path minimising aliasing will be possible, condition 2) grants that such schedule will require less than infinity number of control steps. If the two conditions lead to adding as many functional units as required by the reference architecture described in Section 2, resource sharing is excluded a priori since it would lead to a design aliasing probability that might be higher than in the reference architecture without decreasing its cost.

Let us first describe informally the core concepts of our approach, before presenting the implementing algorithm. We refer from now on to *nominal* and *checking* DFG's thus denoting, respectively, the fully scheduled and allocated DFG corresponding to the nominal data path and the (levelized, but as yet neither scheduled nor allocated) DFG corresponding to the checking data path, that must be scheduled over at most Pk_N control steps. Scheduling and allocation of the nominal DFG are kept unchanged; the problem thus involves scheduling and allocation of the checking DFG only.

We consider a sequence of Pk_N control steps, over which the nominal schedule is repeated P times; information concerning allocation of functional units in the nominal data path is kept available. In any control step c_k ($1 \leq k \leq Pk_N$) all "ready" operations in the DFG are taken into account, an attempt is made to simultaneously schedule and allocate them - satisfying suitably defined priorities - onto available ("free") functional units such that condition a) is satisfied; for all unscheduled operations, priorities are updated and the attempt is repeated in the following control step. (The set of functional units consists of the nominal ones plus, possibly, those inserted to satisfy rules 1 and 2). While availability of a functional unit f_j in control step c_k is determined by examining schedule and allocation in step c_k of the current replica of the nominal schedule, data input to the scheduled operation are always checked against those input to unit f_j in the first iteration of the nominal schedule (i.e., the one on whose results the checking will be performed). This mixed scheduling-and-allocation step is repeated until either the whole checking DFG is scheduled in a satisfactory way over the Pk_N control steps, or no such schedule is found. In the first case, a resource-sharing minimum-cost self-checking solution has been found. Registers for the checking data path are identified and allocated, independently of the nominal ones; the choice of excluding possibility of sharing for the set of registers is justified by the consideration that lifetimes for variables in the checking data path will be usually rather long, so that reusability of registers belonging to the nominal data path would be reduced. Finally, the complete interconnection network is created and the control FSM is synthesised.

If on the contrary no acceptable schedule is found, rather than attempting alternative schedules with the same set of resources we decide to increase such set by adding (one at a time) a functional unit of one of the used types. A heuristic approach is adopted to choose the type of functional unit by which this set is increased, as follows:

I. for each type t_k of functional unit, the number of times scheduling "ready" operations of the corresponding type has been delayed in the checking DFG schedule is counted, and the first control step in which such miss occurred is recorded; the counter is incremented by one for each control step in which a delay occurs, whatever the number of operations delayed;
II. a functional unit corresponding to the highest count and - for equal count value - to the earliest miss is added to the set; in the case of multiple units with the same characteristics, the lowest-cost one is chosen.

This solution attempts to achieve high probability of anticipating the schedule of a larger number of delayed operations without undue increase in computational complexity.

Having thus outlined our general philosophy, we must specify the scheduling-and-allocation technique as well as the priority figure chosen. As regards priority, a widely used figure is *mobility*, which - in the case of time-constrained scheduling - is evaluated for each operation o_i in the DFG as the difference m_i between its ALAP and its ASAP labels [9], priority increasing with decreasing mobility values. We suggest here to use an *extended mobility* evaluated as $e_i=(P-1)k_N+m_i-k_C$, i.e., the original mobility in the nominal DFG increased by the additional number of control steps by which the checking scheduled DFG can extend beyond the nominal scheduled DFG, decreased by the number of control steps required to check results on critical paths. Whenever a ready operation cannot be scheduled in the present control step and has to be delayed, its mobility is decreased by one and so is the mobility of all its (immediate or indirect) successors. Note that whenever an operation o_i such that $e_i=\langle 0 \rangle$ cannot be scheduled in the present control step it can be immediately stated that a schedule within the given time bounds is impossible; thus, failure of the attempt can be declared even before completing analysis of the whole DFG.

The scheduling-and-allocation technique can be summarised as follows:

1) At any given control step c_h, for each type t_k of functional unit determine the set $\mathbf{O}$ of ready operations that can be allocated on units of this type and the set $\mathbf{Fu_k}$ of functional units of type t_k that are not allocated in the nominal DFG during step c_h.
2) Whenever neither $\mathbf{O}$ nor $\mathbf{Fu_k}$ are empty, create a bipartite graph whose nodes represent, respectively, entries in $\mathbf{O}$ and in $\mathbf{Fu_k}$ and a node associated with $o_i \in \mathbf{O}$ is connected by an edge to a node $fu_k^j \in \mathbf{Fu_k}$ iff operation o_i in the nominal DFG has not been allocated to fu_k^j. Edges are labelled with the extended mobility of the operations.
3) A matching is attempted on the bipartite graphs thus created. Whenever a complete matching of operations onto functional units is achieved, all operations are scheduled in the control step and the allocation is given by the matching. Otherwise, an "optimum" non-complete matching is sought, where weights are taken into account. All unmatched operations are delayed to control step c_{h+1}, and the relevant extended mobilities are updated.

Steps 1 to 3 are repeated for increasing values of *h*, until either the whole DFG has been scheduled and allocated or an operation with zero mobility cannot be matched, in which case failure is declared and a renewed attempt is made with suitably increased set of functional units. It may be worthwhile noting that the number of control steps finally required by the checking DFG may be less than Pk_N, since at some control steps the number of "free" resources may be higher than that foreseen by the reference minimum-cost architecture. In any case, checking periodicity will obviously be an integer multiple of k_N.

As an example, we consider the AR filter discussed also in [2,10], whose optimum time-constrained schedule is given in Figure 1.

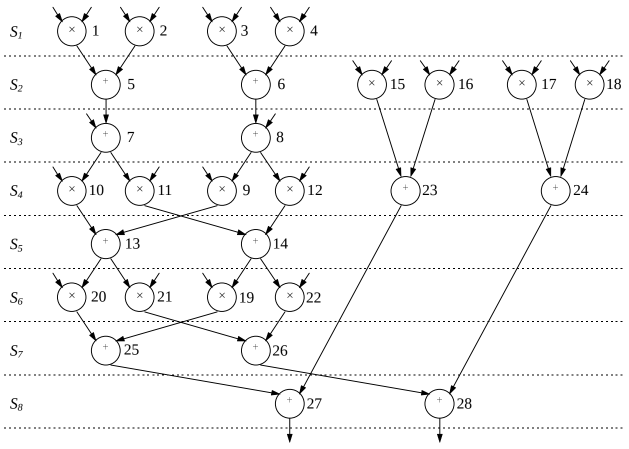

Figure 1: Optimum scheduling for AR filter

A minimum-resource allocation - considering functional units only - involves four multipliers (m_1, m_2, m_3, m_4) and two adders (a_1, a_2) with the following possible binding:

functional unit	operations in the DFG
m_1	*1, 9, 15, 19*
m_2	*2, 10, 16, 20*
m_3	*3, 11, 17, 21*
m_4	*4, 12, 18, 22*
a_1	*5, 7, 13, 23, 25, 27*
a_2	*6, 8, 14, 24, 26, 28*

It is $k_N=8$; we assume as periodicity of checking $P=3$, with no restrictions on the number of checkers (a checking operation requires one control step). Semi-concurrent checking then requires at most 23 control steps to schedule the checking DFG, the 24th step being reserved for checking. The reference architecture is designed accordingly; the DFG can be scheduled over 21 control steps by using just one multiplier, one adder and one checker, i.e., the minimum number of resources.

The algorithm is now applied, attempting first to schedule a checking DFG within 23 control steps, without increasing the set of resources. The attempt fails at step 23, where operations *27c* and *28c* still remain to be scheduled and no free adder is available. Count of delayed types of operations gives 7 delays for multiplications and 20 for additions: adder a_3 is then inserted. The scheduling and allocation algorithm is applied once more, leading to the solution summarised in Figure 2 and in Table 1: where rows correspond to control steps and columns to functional units, entry (a,b) denoting the operation - either of nominal or of checking DFG - performed by functional unit b in control step a; operations belonging to the checking DFG are denoted by the original ordering number followed by the letter *c*.

Table 1: Scheduling and allocation for the self-checking AR filter

control step	m_1	m_2	m_3	m_4	a_1	a_2	$\boldsymbol{a_3}$	***checker***
cs1	1	2	3	4				
cs2	15	16	17	18	5	6		
cs3	***2c***	***1c***	***4c***	***3c***	7	8		
cs4	9	10	11	12	23	24	***5c***	
cs5	***16c***	***15c***	***18c***	***17c***	13	14	***6c***	
cs6	19	20	21	22	***8c***	***7c***	***23c***	
cs7	***10c***	***9c***	***12c***	***11c***	25	26	***24c***	
cs8					27	28	***13c***	
cs9	1	2	3	4			***14c***	
cs10	15	16	17	18	5	6		
cs11	***20c***	***19c***	***22c***	***21c***	7	8		
cs12	9	10	11	12	23	24	***25c***	
cs13					13	14	***26c***	
cs14	19	20	21	22		***27c***	***28c***	
cs15					25	26		***out. 27***
c16					27	28		***out. 28***

It can be noticed that semi-concurrent checking with frequency actually higher than that required can be achieved by introducing just one extra adder and one checker; checking is performed at each *second* iteration of the nominal DFG, results of the checking DFG being available already at the 14th control steps, so that one primary output is checked in the 15th step and the other one in the 16th. Cost and performances are thus better than in the reference architecture.

Some considerations are possible concerning a comparison of costs for the control FSM's in the reference

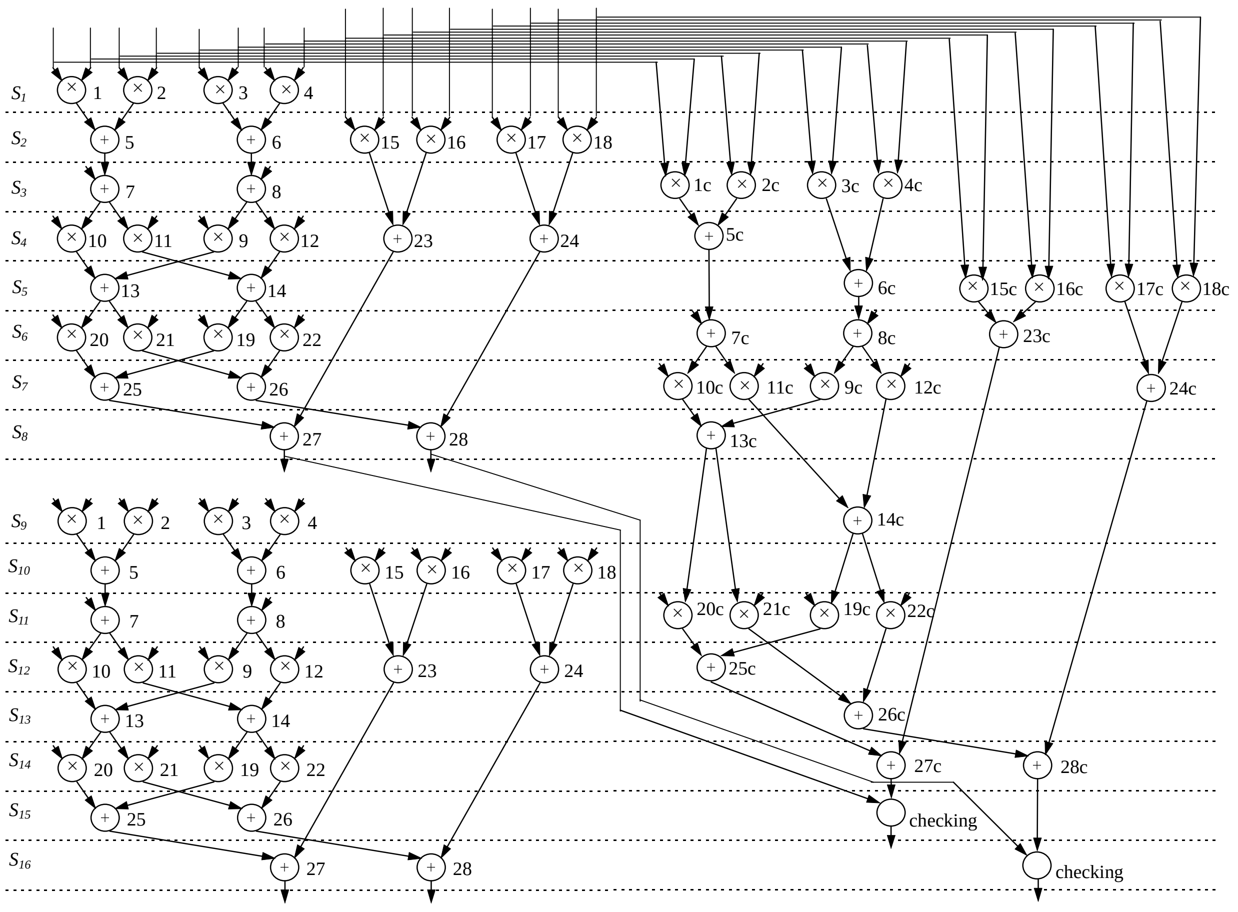

Figure 2: Scheduling for AR filter with semi-concurrent self-checking capability

and in the resource-sharing architectures. Since the number of control steps in both instances is given by design requirements, the number of states of the two machines has the same upper bound; in our example, the cost of the FSM for the resource-sharing architecture will be in effect much lower. As for the width of the *control word*, we should compare the sum width of two control words for the reference architecture with the global width for the resource-sharing one; while no general evaluation is possible, it is conceivable that suitable coding will in general allow more compact control words for the resource-sharing solution.

4: Limits and applicability conditions

As suggested in Section 3, resource sharing between nominal and checking data paths may lead to risk of aliasing; the basic condition guiding allocation of the checking DFG, by which no operation could be allocated to the same functional unit in nominal and checking data path, is only the minimal condition to be satisfied to avoid *certainty* of aliasing on the individual operation.

A path in a DFG can be considered as an ordered sequence of operations. If we examine the scheduling and allocation obtained for a general DFG by means of the algorithm outlined in Section 3, it is quite probable that, referring to a path from primary inputs to a primary (thence checked) output in both nominal and checking DFGs, there will be one or more shared functional units used to implement *different* operations of the same type along the path: this introduces risk of aliasing.

Let us examine the possible instances on a simple sequence of two operations of the same type; results can be applied recursively to more complex sequences.

Let o_1, o_2 be two identical operations constituting a sequence $o_1(o_2(a,b),c))$: assume presence of two identical functional units fu_1, fu_2 whose allocation leads, in the nominal DFG, to $o_1 \Rightarrow fu_1, o_2 \Rightarrow fu_2$, and in the checking DFG to $o_1 \Rightarrow fu_2, o_2 \Rightarrow fu_1$. Assume further that fu_1 is faulty and fu_2 is fault-free. We denote by $f_1^*(x,y)$ the result produced by the faulty unit operating on inputs x,y and by $f_2(x,y)$ the results produced by the fault-free unit operating on the same inputs. Results produced by the allocated operation sequence in the two paths can now be denoted, respectively, as $f_1^*(f_2(a,b),c))$ and as $f_2(f_1^*(a,b),c))$. Aliasing occurs only if $f_1^*(f_2(a,b),c))= f_2(f_1^*(a,b),c))$ and results are affected by an error. Possibilities are as follows:

- $f_2(a,b)= f_1^*(a,b)$: this occurs only if the inputs *a* and *b* do not excite the fault. In this case, results are both error-free (error *masking,* not *aliasing,* occurs). Then:
  - either the pair of inputs $f_2(a,b),c$ excites the fault in fu_1, in which case results produced by the two sequences will be different and no aliasing on the whole sequence is possible, or
  - results of sequence $f_1^*(f_2(a,b),c))$ are affected by error, and different from the ones of sequence $f_2(f_1^*(a,b),c))$: no aliasing occurs;
- $f_2(a,b)\neq f_1^*(a,b)$: inputs *a,b* excite the fault in fu_1. Let us denote $f_2(a,b)=d$, $f_1^*(a,b)=d^*$. Then:
  - if inputs *d,c* do not excite the fault, it is $f_2(d,c)= f_1^*(d,c)$; if $f_2(d,c)\neq f_2(d^*,c)$, the error is detected; otherwise, both results are correct and error *masking* occurs;
  - if inputs *d,c* excite the fault, it is $f_2(d,c)\neq f_1^*(d,c)$; in this case, *aliasing* may occur if $f_1^*(d,c)= f_2(d^*,c)$.

It will be noticed that conditions under which aliasing *might* occur are fairly restrictive and depend on characteristics of the functional units as well as on the specific set of data. Thus, it will be up to the application designer - based on information available for the application as well as for the technological implementation - to evaluate the actual aliasing probability.

5: Extension of the fault assumptions and conclusions

Let us now consider how possible faults affecting *any* component of the Data Path - other than functional units - may be detected by our proposed methodology. We keep the single-fault assumption, extending possibility of errors to *register* and to *switches* in the interconnection network (we assume point-to-point, multiplexer-controlled interconnection networks).

Refer first to faults affecting *register*. Registers are not shared between nominal and checking data paths: thus, a fault affecting a register will affect either the nominal or the checking computation and will be detected. No aliasing is possible; error masking may occur if the data do not excite the fault.

Coming to faults affecting multiplexers, we envision a classical architecture in which multiplexers are present only on inputs of registers and of functional units. If a multiplexer on a register input is faulty, the error can be seen *as if* it were due to the register, and the previous considerations hold. If, on the other hand, the faulty multiplexer is on an input to a functional unit, the consequent error can be seen as affecting a shared resource, and possibility of aliasing (as seen in the previous section) arise.

No proper set of benchmarks is available for the problem here envisioned; we examined some other DSP algorithms, in particular some filters discussed in high-level synthesis literature, results being consistently positive. An interesting example is constituted by the elliptic filter [2], whose nominal (time-constrained) structure requires four adders and operates on eleven control steps. The minimum-resource reference architecture involves one adder and one checkers and requires 27 control steps (26 for the computation and one for checking the last-produced output); thus, checking periodicity of 3 is obtained. A shared-resource solution can be designed *without introducing any further adder*, operating in 16 control steps for the computation proper, so that so that semi-concurrent checking can be achieved with a periodicity of *two*.

References

[1] C. Bolchini, R. Montandono, F. Salice, D. Sciuto: "Self-checking FSM's based on a constant-distance state encoding", *Proc. IEEE DFT 95*, Lafayette, USA, Nov. 1995

[2] A. Orairoglu, R. Karri: "Automatic Synthesis of Self-Recovering VLSI Systems", *IEEE Trans. Comp.*, vol. 45, n. 2, Feb. 1996, pp. 131-142

[3] A.Antola, V.Piuri, M.G.Sami: "Optimising High-Level Synthesis for Self-Checking Arithmetic Circuits", *Proc. IEEE DFT'96*, Boston, MA, USA, November 1996

[4] A.Antola, V.Piuri, M.G.Sami: "A High-Level Synthesis Approach to Optimum Design of Self-Checking Circuits", *Proc. EURODAC 96*, Geneva, Switzerland, September 1996

[5] Y.H. Choi, D.S. Fussel, M. Malek: "Token-Triggered Systolic Diagnosis of Wafer Scale Arrays", *Proc. Int'l Workshop on Wafer Scale Integration*, Southampton, UK, July 1985

[6] R.A. Evans, J.V. McCanny, K.W.Wood: "Wafer Scale Integration Based on Self-Organization", *Proc. Int'l Workshop on Wafer Scale Integration*, Southampton, UK, July 1985

[7] K.D.Wagner, S.Dey: "High-Level Synthesis for Testability: A Survey and Perspective", *Proc. Design Automation Conference 96*, Las Vegas, June 1996, pp.131-136

[8] B.Iyer, R.Karri: "Introspection: A Low Overhead Binding Technique During Self-Diagnosing Microarchitecture Synthesis", *Proc. Design Automation Conference 96*, Las Vegas, June 1996, pp. 137-142

[9] D. Gajski, N. Dutt, A. Wu and S. Lin: *High-Level Synthesis*, Kluwer Academic Publishers, Boston, MA, USA, 1992

[10]G. Buonanno, M. Pugassi, M.G. Sami: "A High-Level Synthesis Approach to Design of Fault-Tolerant Systems", *Proc. 1997 IEEE VLSI Test Symposium*, Monterey, CA, USA, Apr. 1997