

Semi-Concurrent Error Detection in Data Paths

Anna Antola, Vincenzo Piuri, Mariagiovanna Sami
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza L. Da Vinci 32, 20133 Milano, Italy
e-mail: {antola,piuri,sami}@elet.polimi.it

Abstract

An innovative approach for high-level synthesis of digital circuits with semi-concurrent self-checking abilities is introduced, achieving a compromise between redundancy and checking effectiveness. Attention is mainly focused on the data path, described as a general Sequencing Graph including linear paths as well as loops and branches. A reference architecture is defined; a technique allowing to reduce redundancy through resource sharing is then introduced, leading to synthesis of the self-checking architecture. An algorithm is proposed to simultaneously schedule and allocate the resources, while keeping error aliasing as reduced as possible. The desired checking periodicity is guaranteed by the algorithm.

1. Introduction

A typical approach in designing integrated application-specific systems is nowadays based on the use of high-level synthesis techniques: from the behavioural description of the system and from the algorithmic definition of the computation, a graph description can be derived to capture both the data and the control flows in the so-called Sequencing Graph (SG) [5]. From the analysis of such graph and the application constraints, it is possible to identify the resources required to perform the desired computation as well as to obtain optimum scheduling, allocation and binding [5,9]. The resulting architecture is composed of a data path and its controlling FSM; typically, it is evaluated with respect to latency, throughput and resources, according to the criticality of these figures of merit for the envisioned application.

In several mission-critical environments (e.g., in telecommunications and in automotive electronics), the presence of self-checking features is highly desirable; on the other hand, cost and power consumption make it necessary to avoid straightforward redundancy. Solutions based on suitable coding allow to obtain a concurrently self-checking FSM (e.g., see [1]). Coding cannot in general be immediately applied to the data path, since it would easily lead to excessive increase of resources (checkers) and latency (due to increase of the control step to accommodate checking); moreover, in the most general case, the various operations in the data path would require different codes.

In [2,3], it was suggested to introduce the self-checking requirements directly from the initial steps of high-level synthesis, modifying the synthesis approach so as to optimise area and performances while granting autonomous error detection; a *concurrent* error detection by means of a code-based solution valid for Data Flow Graphs consisting of *arithmetic operations* only

was discussed. In [7], a technique granting *fault location* is proposed by exploiting, at each control step, the presence of unused non-redundant resources to repeat operations in the DFG; the null-redundancy requirement (at least, as far as functional units are concerned) leads to the drawback that for some operations neither checking nor location are feasible.

In several cases, actually, fault statistics as compared with an ASIC's latency are such that totally concurrent self-checking is by far too costly – in terms of resources and performances – with respect to fault detection and confinement requirements. More specifically, the worst-case time interval between two subsequent faults is (even much) greater than the time required to execute the ASIC's process, and even then it might be acceptable to propagate a limited number of erroneous results provided a pre-defined upper bound was not overcome (this is typical of applications with inherent data redundancy, as in many signal and image processing applications).

Then, so-called *semi-concurrent* techniques become a viable solution. In these approaches, the nominal computation is never suspended: checking is performed on-line, periodically, in parallel with the nominal computation, and on a set of input data extracted from the nominal data flow at the beginning of every checking period. Thus, only the first execution of the nominal algorithm is verified in the checking period: the subsequent executions within the period remain unchecked. As a consequence, erroneous results, produced by the nominal algorithm in the unchecked executions due to a fault occurring at this time may be delivered before the error status is asserted; a "latency" of fault detection is introduced, and its acceptable bound concurs to determine checking periodicity. The architecture performing the nominal computation is unchanged, so its performances are not affected and a very limited number of resources is added to the nominal system. Semi-concurrent self-checking solutions have been advocated in the past for regular, iterative architectures (e.g., see [4,6]), where low redundancy was granted by introducing a limited number of spare modules in the nominal architecture.

In the present paper, we propose the use of a *semi-concurrent* self-checking technique in high-level synthesis of an arbitrary application-specific system, described by a Sequencing Graph (SG) [5], on which no constraints are given either in terms of structure or in terms of allowable operation types. The innovative aspect consists of taking into account the self-checking problem directly from the beginning of the high-level synthesis steps and in incorporating these self-checking features and strategy into the high-level design methodology. We concentrate on data path checking; our goal is to synthesise a modified data path that allows semi-concurrent self-checking with minimum redundancy (in terms of functional units) and minimum risk of aliasing for the adopted periodicity of checking.

In Section 2, we introduce the fault assumptions and the approach to semi-concurrent self-checking circuit design. In Section 3, we present the basic algorithm for simultaneous scheduling and allocation in the case of a linear SG. Section 4 extends the technique to deal with SG containing loops, while the case of branches is discussed in Section 5. The effectiveness of the approach is shown through some examples.

2. The semi-concurrent self-checking architecture design

The *nominal architecture* is the one initially designed *without* any self-checking capacity: scheduling, allocation and binding are performed on the basis of cost and performance. We assume here latency as the primary figure of merit, but this does not constitute a restriction for our approach. Semi-concurrent self-checking features are then provided by introducing the *checking architecture* so as to perform the redundant computation verifying at least one nominal execution within each checking period: a *self-checking architecture* is thus obtained.

Required *checking periodicity* C_p is defined on the basis of application requirements and fault rate; latency of the checking architecture (including checking of the results) must then be $l_c = k_c \tau \leq C_p$, where k_c is the number of control steps required by the scheduling and τ is the length of the clock cycle. Let $l_N = k_N \tau$ be the latency of the nominal architecture: the *relative checking periodicity* P is defined as $P = \lceil l_c / l_N \rceil$.

Operation of the *checking architecture* is as follows:

1. The same set of nominal data (*checked data*) is fed to both nominal and checking circuits and the two circuits operate separately. The primary outputs produced by the nominal circuit are stored until the corresponding values have been computed by the checking circuit; then checking is performed by *self-checking checkers*;
2. After Pk_N control steps, either an error has been detected or both circuits are declared fault-free; at this time, both circuits receive a new set of checked data and step 1 is repeated.

The *self-checking architecture* is designed starting from the SG of the nominal circuit and implementing the lowest-cost data path compatible with the checking periodicity. In general, in the nominal circuit not all functional units present will be actually used in *each control step*; therefore, a low-cost self-checking data path can be designed by scheduling and allocating both nominal and checking data path onto the same *shared* resources. The nominal data path is synthesised, its set of registers is increased and its interconnection structure is modified to deal with the checking data path, but without increasing the set of functional units or by adding the minimum number of such units. For simplicity's sake, registers are not shared between the nominal and the checking architectures.

The following necessary conditions must be verified:

- a) the probability of aliasing is as low as possible;
- b) scheduling of the checking SG onto the shared resources must be possible within the periodicity P evaluated for the envisioned application.

A minimum requirement is that *no operation is performed by the same functional unit both in nominal and checking SGs*. A preliminary analysis of the nominal data path leads to possible increase of the functional units:

- 1) whenever for a given type of functional unit, one instance only is present in the nominal data path, a second instance must be added to satisfy condition a);
- 2) whenever for a given type 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), so that the checking data path needs a finite number of control steps.

To design the checking architecture and to evaluate the detection capabilities, we adopt the following fault assumptions:

- faults are concentrated in *functional units*; registers and connection network are assumed fault-free for presentation simplicity. Such restriction can then be relaxed as discussed in Section 6, leading to a more comprehensive fault model;
- at most a *single fault* 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.

3. The basic approach for a linear SG

Let's consider first the simple case of a SG without branches and loops. At first, we introduce informally our approach to mixed scheduling and allocation: more details are given in [8]. We refer from now on to *nominal* and *checking* SG's denoting, respectively, the fully scheduled and allocated SG corresponding to the nominal data path and the (levelised, but as yet neither scheduled nor allocated) SG corresponding to the checking data path.

We consider a sequence of control steps, over which the nominal schedule is repeated P times. In any control step c_h , all "ready" operations in the checking SG are taken into account to simultaneously schedule and allocate them onto unused functional units according to given priorities, such that condition a) is satisfied; for all unscheduled operations, priorities are updated and the attempt is repeated in the following control step. While availability of a functional unit $u_{h,k}^j$ of type j in control step c_h is determined by examining schedule and allocation in control step c_h of the current replica of the nominal schedule, data input to the scheduled operation are always checked against those input to unit $u_{h,k}^j$ in the first iteration of the nominal schedule. This mixed scheduling-and-allocation step is repeated until either the whole checking SG (including the checker) is scheduled within Pl_N or no such schedule can be 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 scheduling-and-allocation algorithm can be summarised as follows:

1. At any given control step c_h , for each type t_k of functional unit determine the set $\mathbf{O}_{h,k}$ of ready operations that can be allocated on units of this type and the set $\mathbf{U}_{h,k}$ of functional units of type t_k that are not allocated in the nominal SG during step c_h .
2. Whenever neither $\mathbf{O}_{h,k}$ nor $\mathbf{U}_{h,k}$ are empty, create a bipartite graph whose nodes represent, respectively, entries in $\mathbf{O}_{h,k}$ and in $\mathbf{U}_{h,k}$ and a node associated with $o_{h,k}^i \in \mathbf{O}_{h,k}$ is connected by an edge to a node $u_{h,k}^j \in \mathbf{U}_{h,k}$ iff operation $o_{h,k}^i$ in the nominal SG has not been allocated to $u_{h,k}^j$. Each edge is labelled with the operation's *extended mobility* defined as $e_{h,k}^i = (P-1)k_N + m_i - k_C$, i.e., the traditional mobility m_i (given by the difference between its ALAP and its ASAP labels [9]) in the nominal SG increased by the additional number of control steps by which the checking scheduled SG can extend beyond the nominal scheduled SG, decreased by the number of control steps required to check results on critical paths.
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 related extended mobilities are updated.

Steps 1 to 3 are repeated for increasing values of h , until either the whole SG has been scheduled and allocated or an operation with zero mobility cannot be matched. In this last case, failure is declared and a renewed attempt is made with suitably increased set of functional units: we increase the set of resources by adding (one at a time) a functional unit of one of the used types, so as to anticipate the schedule of a larger number of delayed operations without undue increase in computational complexity.

It may be worthwhile noting that the number of control steps finally required by the checking SG 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 aliasing is concerned, step 2 is a necessary and sufficient condition to avoid aliasing in the individual operation only. If a functional unit is used to implement different operations within the

same sequence of operations, additional conditions need to be defined to prevent aliasing, as discussed in [8].

4. The approach for general SG graphs: the case of loops

Let us consider how the technique introduced can be extended to support SG's including loops. We restrict our analysis to the case of *data-independent* numbers of iterations; otherwise, complexity of the controlling FSM explodes, so that it is more reasonable to envision - for such instances - self checking performed by means of conventional redundancy techniques. We discuss here the case of a single loop; extension to multiple and/or nested loops does not require further theoretical analysis.

Referring to the SG abstraction in figure 1, H is the *head section*, B is the *loop body* and T is the *tail section*. The nominal schedule requires N_H control steps for the head, N_B for the body, and N_T for the tail. The loop body is iterated K times in any execution instance. In a similar way, assuming that a schedule has already been created for the checking SG, we define C_H , C_B and C_T . The total latency k_N (in number of steps) for the nominal SG is $k_N = N_H + KN_B + N_T$, while it is $k_C = C_H + KC_B + C_T$ for the checking SG. As for linear SG's, we assume that the optimum schedule-and-allocation choice for the nominal SG is kept unchanged when scheduling and allocation is performed for the checking SG.

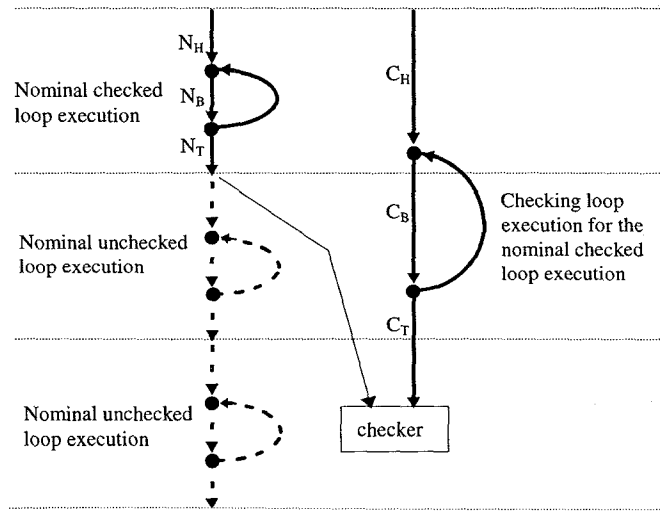


Figure 1: Nominal and checking data paths with loop.

A trivial solution consists in complete unrolling of both **nominal** and checking SG (possible consequent optimisations, if any, must be identical for both); in this case, the basic algorithm is applied in a straightforward way. While possibly leading to lower latency, this leads to a larger number of *control words* for the controlling FSM.

Let us then examine the alternative approach, by which loops are kept unchanged. Assume that control step c_h has been reached during scheduling and allocation of checking SG. $c_h^* = c_h - 1 \bmod k_N$ gives the execution step of the nominal SG in which c_h is placed. The following holds:

- if $1 \leq c_h^* \leq N_H$, selection of the operations in the checking SG to be scheduled in control step c_h and related allocation is performed as for a linear SG;
- if $N_H+1 \leq c_h^* \leq N_H+KN_B$, the operations reached by execution of the cycle in the nominal SG is determined as $c_h - N_H - 1 \bmod N_B$, and this information is used for the checking data-path, as above;
- if $N_H+KN_B+1 \leq c_h^* \leq k_N$, the basic algorithm is again applied.

5. The approach for general SG graphs: the case of branches

Consider now the presence of *branches* in the SG, as shown in figure 2. Denote by E_N a checked execution of the nominal SG and by E_C the corresponding execution of the checking SG. While it is obvious that the same branching alternative is taken in both E_N and E_C , no comparable assertions can be made for alternatives taken in E_C and in the executions of the nominal SG between E_N and the subsequent checked execution E_{N+1} . This must be taken into account when designing the checking data-path.

Let k_N be the latency of the nominal SG, and c_h be the control step reached in design of the checking data-path. Consider for simplicity a two-way branch without any other mandatory data-path to be executed in parallel to the branch (as shown in figure 2); we denote by A and B the two alternatives.

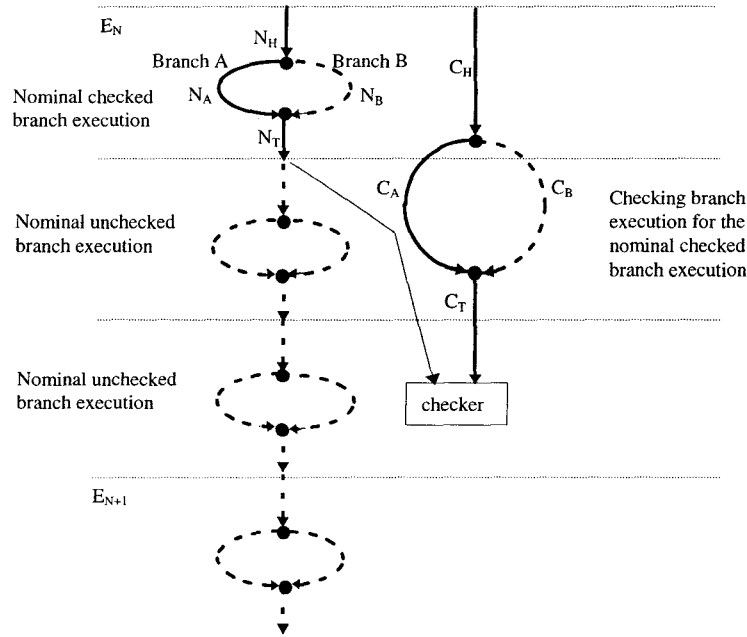


Figure 2: Nominal and checking data paths with branches.

The basic algorithm presented in Section 3 is modified to guarantee availability of resources independently from the alternative actually performed in each iteration of the nominal and

checking executions; in particular, operations are extended to take into account the possible alternatives insofar as creation of the bipartite graph is concerned. More specifically:

- for any step c_h , if both scheduled operations of the nominal SG and ready (but unscheduled) operations of the checking SG are external to the branching construct, the basic algorithm is applied.
- for any step c_h , if the scheduled operations of the nominal SG are external to the branching construct but the ready (and unscheduled) operations of the checking SG are internal to the branching construct, a bipartite graph is built separately for each of the branches in the checking SG as in the basic algorithm. Scheduling and allocation are performed by the basic algorithm for each branch independently.
- for any step c_h , if the scheduled operations of the nominal SG are internal to a branching construct but the ones of the checking SG are not, the set of available functional units $U_{h,k}$ is obtained by removing the functional units $u_{h,k}^j$ allocated in *any* branch of the nominal SG at that step (this extends the definition of Step 2 in the basic algorithm). Scheduling and allocation are then performed on these sets by using the Step 3 of the basic algorithm.
- for any step $1 \leq c_h \leq k_N$, if operations of both nominal and checking SG fall within the branching construct, two bipartite graphs are built by extending the Step 1 of the basic algorithm, namely:

1. one bipartite graph including operations of branch A_C and functional units $u_{h,k}^j$ not used by branch A_N in the same step c_h ,
2. a corresponding one for branches B_C and B_N .

Steps 2 and 3 of the basic algorithm are then separately applied to each of these bipartite graphs.

- for any step $k_N \leq c_h \leq k_C$, if operations of both nominal and checking SG fall within the branching construct, one bipartite graph is built for each branch of the checking SG as follows, by extending the Step 1 of the basic algorithm:

1. the set of available functional units in either graph ($U_{h,k}^A$ and $U_{h,k}^B$) is obtained by removing the functional units $u_{h,k}^j$ allocated in *any* branch of the nominal SG,
2. operations $o_{h,k}^j$ of one branch only of the checking SG are inserted.

Steps 2 and 3 of the basic algorithm are then separately applied to each of these bipartite graphs; conditions to avoid aliasing in Step 2 must refer to the operations in the nominal execution under checking.

Since Step 2 is performed as in the basic algorithm discussed in Section 2, the same conditions concerning aliasing for individual operations and sequences of operations hold in SG with branches as well.

To reduce the error masking probability, the functional units, that can be used in both branches A and B at control step c_h (i.e., that belong both to $U_{h,k}^A$ and $U_{h,k}^B$) as well as at control step c_{h+a} (i.e., that belong both to $U_{h+a,k}^A$ and $U_{h+a,k}^B$) and that have been used to allocate operations in branch A at control step c_h , should be used to allocate operations in branch B at control step c_{h+a} , and vice versa.

To show clearly the application of this technique, we consider the simple SG in figure 3a, having empty head and tail sections, with two alternative branches (A and B) and without any other subgraph of the SG being in parallel with such branches. Scheduling of the nominal SG by a traditional time constrained approach is shown in the leftmost part of figure 3b; both alternatives must be available so that the branch actually executed will be chosen according to the current value of the branch selector. The scheduled SG requires 3 control steps to be

completed. Allocation of the nominal SG can be performed as shown in table 1, by using 2 multipliers, 2 adders, and 1 subtractor. The branch executed is data dependent and may change for each execution. The checking SG built according to our modified algorithm for minimum hardware redundancy is shown in the rightmost part of figure 3b, while the resulting allocation leading to a low aliasing probability is given in table 2. The scheduled checking SG requires one additional subtractor to avoid aliasing and the checker; generation of the checking result is performed in 5 control steps, while the checker can operate in the sixth control step allowing for a relative checking periodicity P equal to 2.

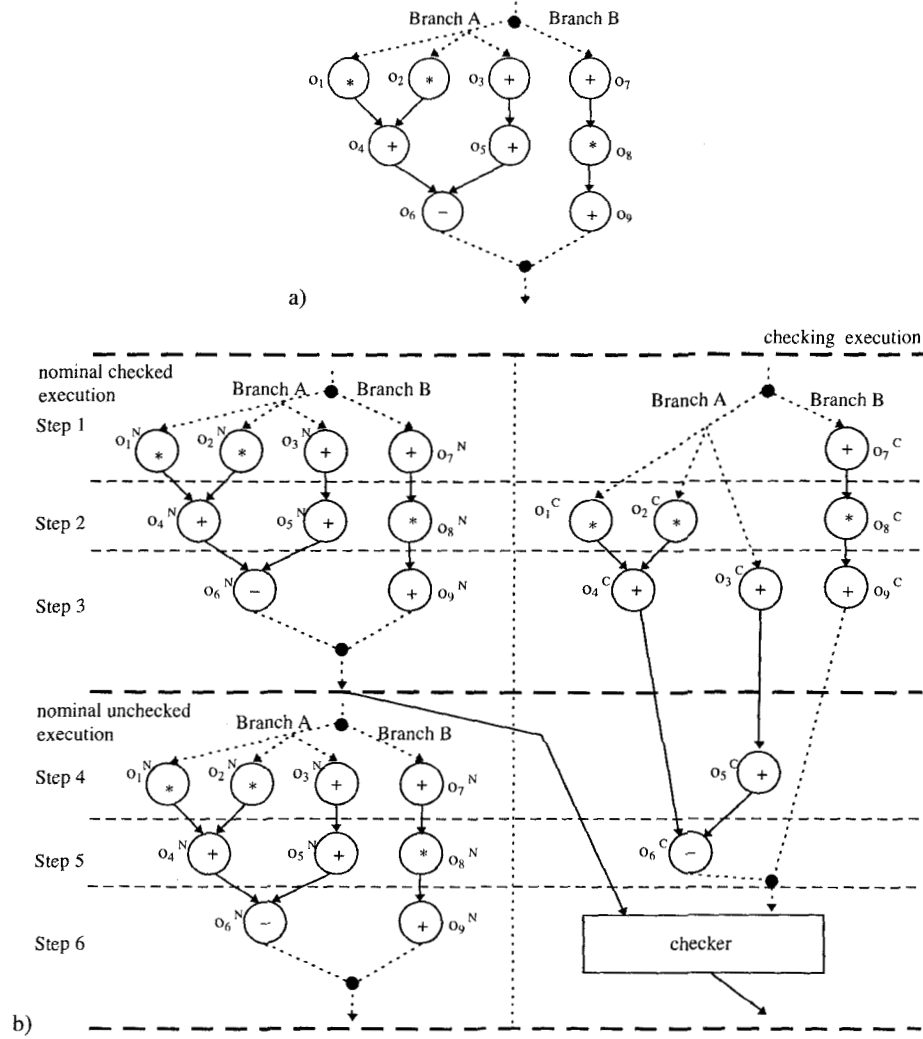


Figure 3: Application of the modified algorithm for data paths with branches: the SG (a), the scheduled nominal and checking SGs (b).

Table 1: The allocation of the nominal SG for the example of Figure 3.

Operation	o_1^N	o_2^N	o_3^N	o_4^N	o_5^N	o_6^N	o_7^N	o_8^N	o_9^N
Unit	m_1	m_2	a_1	a_2	a_1	s_1	a_1	m_1	a_1

m_1, m_2 : multipliers; a_1, a_2 : adders; s_1 : subtractor.

Table 2: The allocation of the checking SG for the example of Figure 3.

Operation	o_1^C	o_2^C	o_3^C	o_4^C	o_5^C	o_6^C	o_7^C	o_8^C	o_9^C
Unit	m_2	m_1	a_2	a_1	a_2	s_2	a_2	m_2	a_2

m_1, m_2 : multipliers; a_1, a_2 : adders; s_1, s_2 : subtractors.

6. Conclusions and extension of the fault assumptions

In this paper, we presented a novel approach to design of self-checking data paths by introducing semi-concurrent checking as a feature of high-level synthesis. An algorithm is presented for simultaneous scheduling and allocation at the minimum additional costs in terms of resources with respect to the nominal architecture designed by satisfying the nominal constraints on resources and latency. The case of linear SG is extended to deal also with branches and loops.

In the analysis here presented, faults were assumed to affect only the functional units. Keeping the single-fault assumption, we can extend our approach to deal with faults affecting *any* component of the data path in a straightforward way. We consider point-to-point, multiplexer-controlled interconnection networks in the data path.

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 *switching 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 sections) arises.

References

- [1] C. Bolchini, R. Montandon, 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. Antola, V. Piuri, M.G. Sami: "Optimising High-Level Synthesis for Self-Checking Arithmetic Circuits", *Proc. IEEE DFT 96*, Boston, MA, USA, November 1996
- [3] 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
- [4] 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
- [5] G. De Micheli: *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, NY, 1994
- [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] 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
- [8] A. Antola, V. Piuri, M.G. Sami: "A Low-Redundancy Approach to Semi-Concurrent Error Detection in Data Paths", Politecnico di Milano, Dept. of Electronics and Information, Int. Rep. 97-030, Milano, Italy, March 1997
- [9] D. Gajski, N. Dutt, A. Wu and S. Lin: *High-Level Synthesis*, Kluwer Academic Publishers, Boston, MA, USA, 1992