Semiconcurrent Error Detection in Data Paths

Anna Antola, *Member*, *IEEE*, Fabrizio Ferrandi, *Member*, *IEEE*,
Vincenzo Piuri, *Senior Member*, *IEEE*, and Mariagiovanna Sami, *Senior Member*, *IEEE*

Abstract—A high-level synthesis strategy is proposed for design of semiconcurrently self-checking devices. Attention is mainly focused on data path design. After identifying the reference architecture against which cost and performance are evaluated, a simultaneous scheduling-and-allocation strategy is presented for linear-code data flow graphs, allowing resource sharing between nominal and checking data paths. The proposed strategy is actually independent from a specific scheduling-and-allocation algorithm since it is essentially concerned with the introduction of the fault tolerance issue at high-abstraction level in any design environment. Conventional duplication with comparison, even if considered in a high-level synthesis strategy, leads to high circuit complexity increase. The proposed approach provides that the required checking periodicity is satisfied while minimizing additional functional units by means of maximum reuse of the resources available for the nominal computation as long as error detection ability is preserved. The strategy is then extended to deal with branches and loops in the data path. Risk of error aliasing due to resource sharing is analyzed.

Index Terms—Semiconcurrent error detection, self-checking circuits, checking periodicity, fault tolerance, high-level synthesis, data flow graph, resource minimization.

1 Introduction

Traditional figures of merit for ASIC design—cost (expressed as silicon area), latency (expressed as the time required to generate an output), throughput (expressed as the number of outputs delivered in the time unit), etc.—must of necessity be completed with evaluation of testability and cost of testing. Algorithm-specific architectures seldom inherently exhibit those characteristics (such as regularity, ease of controllability and observability, possibility of partitioning, etc.) that facilitate testing. Adding design-for-testability features to a complete logic-level design may become excessively costly in terms of both area and performance. In recent years, a number of authors have advocated introduction of test-related techniques since the first synthesis steps. In particular, solutions aiming at high-level synthesis have been proposed [1], [2]; in the same line, introduction of BIST features allows us to achieve autonomous testing [3]. BIST techniques correspond to an off-line testing philosophy, suitable for end-of-production testing or for periodic life-time testing.

For demanding applications, requiring high *reliability* of the digital system and *correctness* of the results, some form of on-line testing must be introduced. High-level synthesis approaches leading to concurrent self-checking or even fault-tolerant systems have recently been introduced. Such solutions allow to exploit characteristics of the *application* (as specified by the algorithm implemented by the module) to achieve the required performance while limiting redundancy. Thus, in [4], [5], [20], the problem of autonomous error detection and recovery from *transient* faults was in particular taken into account.

Referring to *permanent* faults, a *concurrent* error detection technique based on use of arithmetic codes was presented in [6], [7] for Data Flow Graphs (DFGs) [11] consisting of *arithmetic operations* only. Scheduling and allocation algorithms proposed there provide detection of any single fault (within a fairly comprehensive fault model) with low error latency (i.e., the time between the error occurrence and its detection) and limited area redundancy, checking being performed at suitable intermediate points (as well as on primary outputs) so as to minimize the number of checkers while avoiding aliasing. In [8], a technique providing *fault location* is proposed by exploiting, at each control step, unused nonredundant 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 units neither checking nor location are feasible.

In self-checking solutions—including the ones based on duplication with comparison—[4], [6], [7], [19], [20], checking is applied to the results produced either by the complete DFG or by some suitably defined segments of the DFG itself. Concurrency of checking thus refers to the complete process or to segments of the process, not to individual operations (as is done in [8]); at the end of every iteration of the process implemented by the ASIC, correctness of its outputs (and, possibly, of some intermediate results) is verified. Complete duplication of the hardware resources leads to high circuit complexity.

In the present paper, we propose to introduce a high-level synthesis approach supporting *semiconcurrent* self-checking. The rationale justifying adoption of semiconcurrent testing derives from the low fault occurrence rates of present silicon technologies that—excepting the case of extremely severe operation environments—make it acceptable to perform checking operations not concurrently with

- *A. Antola, F. Ferrandi, and M. Sami are with the Department of Electronics and Information, Politecnico di Milano, piazza Leonardo da Vinci 32, 20133 Milano, Italy.*
  E-mail: {antola, ferrandi, sami}@elet.polimi.it.
- *V. Piuri is with the Department of Information Technologies, University of Milan, via Bramante 65, 26013 Crema (CR), Italy.*
  E-mail: vincenzo.piuri@unimi.it.

Manuscript received 28 Aug. 1998; revised 21 Nov. 2000; accepted 6 Feb. 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 106011.

each process iteration, but periodically, in correspondence of each Nth process iteration (N being suitably defined with respect to the application and the expected fault occurrence rate). Semiconcurrent techniques are effective for systems that operate on a continuous flow of data and that cannot be excluded from nominal operation in order to undergo an off-line test phase. The flow of input data can be considered as a long sequence of random test vectors, leading to acceptable fault coverage. Such techniques have been proposed in the past for regular architectures—typically, linear arrays or rectangular arrays [9], [10]—where a limited number of redundant units cyclically duplicate operation of nominal units so as to achieve fault location as well as fault detection. As in concurrent self-checking, *nominal* input data are used for checking purposes; error latency is thus due not only to checking periodicity but also to possible nonexcitation of existing faults by the present input data. The results achieved by these approaches thus offer a compromise between cost (expressed in terms of area redundancy) and checking efficiency (relating mainly to error latency).

The semiconcurrent self-checking approach proposed in the present paper relates to arbitrary application-specific systems, described by DFGs [11]. Since more general computation structures may incorporate alternative execution paths (branches) as well as iterative execution paths (loops), we will extend our approach to deal with the linear parts of these computation graphs. No constraints are given on operation types. The aim of our approach can be summarized as follows:

> for a given optimum nominal scheduling and allocation of a DFG, and for a given *periodicity of checking*, synthesize a modified data path that will allow semiconcurrent 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. Some preliminary results were published in [12], [13].

The innovation described in this paper is essentially the introduction of the semiconcurrent self-checking strategy in the high-level synthesis environment. Our goal is to show that this approach can be effectively used in high-level synthesis and that a smaller redundancy is necessary than in the conventional case of full duplication with comparison since hardware resources are reused as much as possible whenever the self-checking ability is not impaired. In particular, we are neither proposing a new allocation and scheduling algorithm nor focusing on a specific existing algorithm since the basic idea can be introduced in any of these algorithms. For clearness' sake, we therefore adopt a very basic algorithm. Obviously, more efficient scheduling and allocation algorithms will lead to maximum exploitation of the hardware resources introduced for the nominal computation. The overhead required by self-checking in the semiconcurrent perspective will be less anyway than the conventional duplication in comparison to that which is adopted in the solutions currently available in the literature.

In Section 2, we define the fault model and the rules by which periodicity of checking is computed, as well as the reference architecture against which results produced by our synthesis technique is evaluated. The high-level synthesis approach for minimum-cost semiconcurrent self-checking data paths is defined in Section 3 for linear-code DFGs; in Section 4, the algorithm is extended to deal with branches and loops. In Section 5, the problem of aliasing is examined and the guidelines by which the application designer can evaluate the risk of aliasing for the specific application are introduced. Effectiveness of the proposed strategy is discussed in Section 6, where experimental evaluation is presented.

2 Fault Assumptions and Reference Architecture

The approach chosen here to achieve semiconcurrent self-checking is to modify standard high-level synthesis techniques so that the synthesized data path has self-checking properties. The fault model adopted is therefore of necessity a *functional* one that is technology-independent. We assume that:

1. Faults affect *functional units*, *registers*, and point-to-point *interconnection networks*. Usually, other authors (see, e.g., [4], [5], [8], [20]) assume that faults are only in functional units because of their relative complexity with respect to switches and registers.

   In our perspective, faults in registers are viewed and treated as equivalent to faults in functional units connected to such registers. In fact, contents of a faulty register are either outputs that are subject to periodic checking or inputs to functional units. In the first case, if the data excite the fault, checking will generate an error signal. In the second case, an error-affected operand will be fed to a fault-free functional unit so that an error will appear in the results and will ultimately be detected as if it were due to a faulty functional unit.

   Similarly, faults in point-to-point interconnection networks are viewed as faults in suitable functional units. As a consequence of a single fault located in a multiplexer, either a functional unit will be fed with a "wrong" operand or a "wrong" result will be stored in a register. The fault may affect either the multiplexer's addressing function (an unwanted data input is routed to the multiplexer's output) or data transferred (e.g., a stuck-at on an output line). These errors will be seen from the checking structure as if they affected the functional unit connected with the faulty multiplexer.

   Since faults in registers and interconnections can be dealt with as faults in suitable functional units, for simplicity's sake, but without loss of generality, in the sequel we will refer explicitly only to faults in this last kind of hardware resources.
2. At most a *single fault* (i.e., a single faulty hardware resource, either unit, register, or interconnection) 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 semiconcurrent self-checking operations in order to make the assumption acceptable. In the case of very complex systems, it is also possible to partition the system into subsystems, each individually provided with self-checking capacities, thus allowing for presence of a larger number of faults.

We now introduce a *self-checking reference architecture* against which architectures designed by means our high-level synthesis approach will be evaluated.

We denote as *nominal architecture* the architecture implementing the nominal data path and designed *without* any self-checking capacity; scheduling and allocation are performed on the basis of cost and performance requirements only by using any scheduling-and-allocation algorithm (i.e., the one preferred by the designer). We adopt in this paper 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: This does not in any way constitute a restriction for our approach. Let τ denote the length of the clock cycle: The latency of the nominal architecture is $l_N = k_N\tau$. The *checking periodicity* is defined as the maximum time T which is allowed between two successive checking actions. Actually, we refer to the *relative checking periodicity*, P, computed as the ratio $P = \lfloor \frac{T}{l_N} \rfloor$; if $T < l_N$, the system is partitioned into subsystems, each with latency suitably lower than T, and the approach is applied to each subsystem individually.

An *independent checking architecture* is now designed, starting from the DFG of the nominal process and implementing the *lowest-cost data path compatible with P*. If fault information allows it (i.e., if $T >> l_N$), 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 as allowable latency and cost as the secondary figure of merit will be adopted. Let $l_C = k_C\tau$ be the latency of the circuit thus designed; the time $l_{CHECKING} = k_{CHECKING}\tau$ required for checking results on the critical paths must be added to obtain latency of actual checking. Periodicity of checking is satisfied if $T \geq l_C + l_{CHECKING}$. If $P^* = \lceil \frac{l_C + l_{CHECKING}}{l_N} \rceil < P$, then we can use P^* as checking periodicity and achieve a more frequent checking without increasing the hardware resources. Assuming that a checker operates in one control step, $k_{CHECKING}$ depends only on the chosen schedule and on the number of checkers available.

The *self-checking reference architecture* is obtained by composition of the nominal and the independent checking architectures, and its operation is as follows:

1. At the beginning of a checking cycle, the same set of nominal data (*checked* data) is fed to both nominal and checking architectures. The two architectures then operate independently; in particular, after k_N control steps, the nominal architecture will receive a new set of input data and start operating on them. The primary outputs produced by the nominal architecture operating on the checked data are stored in buffer registers until the corresponding values have been computed by the checking architecture; then, checking is performed and the buffer registers in the nominal architecture are freed. *Self-checking checkers* are used for checking; the number of checkers depends on the DFG structure as well as on latency and cost requirements;
2. After $k_C + k_{CHECKING}$ control steps, either a checking operation has detected an error and an error signal is activated or both architectures are declared fault-free; after P iterations by the nominal architecture, both architectures receive a new set of checked data and Step 1 is repeated.

The two architectures 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 architectures 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 architecture. If a minimum-cost checking data path has been designed, in particular, we need only add one to each type of functional unit required by the nominal data path. A rough evaluation of the controlling FSMs' complexity identifies a "checking" FSM with $k_C + k_{CHECKING}$ states and a simple controller that ensures that both data paths are fed the same data after each sequence of $P \cdot k_N$ control steps.

In general—as noticed in [8]—in the nominal data path, not all functional units present will be actually used *in each control step*; based on this consideration, we explore the possibility of reusing the functional units of the nominal architecture in the checking one 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 data path sharing the nominal data path's resources will be presented.

3 Resource Sharing for Minimum-Cost Semiconcurrent Self-Checking of Data Flow Graphs

We introduce our high-level synthesis strategy by referring initially to the case of DFGs—such that neither branching nor loop constructs are included. To reduce cost of the semiconcurrent self-checking data path, we envision the possibility of sharing some resources of the nominal data path with the checking data path. More precisely, we first schedule and allocate the nominal data path by using any scheduling-and-allocation algorithm (the designer can choose the preferred one, without any restriction). The resulting nominal architecture as well as the related schedule and allocation are frozen and never changed during the subsequent construction of the checking data path. If feasible, the checking data path is then scheduled

and allocated within the given time constraint defined by the checking periodicity and by introducing as few additional resources as possible exceeding the ones of the nominal architecture. If a resource of the nominal architecture is not used in a step and could be used by the checking data path without aliasing impairing the detection abilities, such a resource becomes shared between the nominal and the checking data paths. Otherwise, an additional functional unit is introduced for the checking data path. The set of registers is increased and the interconnection structure is modified as required by the mapping of the checking architecture.[1] To minimize the addition costs, a set of necessary conditions must be verified first:

1. As a necessary condition for avoiding aliasing, we must ensure that *no individual operation in the DFG will be performed in both nominal and checking data path by the same functional unit*. This implies restrictions on scheduling and allocation of the checking DFG;
2. Scheduling of the checking DFG onto the shared resources must be possible within the given periodicity P.

Condition 1 ensures that it will be possible to achieve proper scheduling and allocation of the self-checking data path with minimum risk of aliasing. Condition 2 ensures that such a schedule will require only a finite number of control steps. A preliminary analysis of the nominal data path may lead to an immediate increase of the set of functional units. An instance of a given type of functional unit must be added when at least one of the following conditions holds:

1. whenever one instance only of that type is present in the nominal data path (to achieve Condition 1);
2. whenever all instances of that type are used in each control step of the nominal schedule (to achieve Condition 2).

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 whose aliasing probability 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 implementation algorithm. We refer from now on to *nominal* and *checking* DFGs, 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 which 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. In the sequel, superscript N and C for a given operation o_j will denote, respectively, the appearance of the operation in the *nominal* or in the *checking* DFG.

1. We choose to avoid sharing of registers between nominal and checking datapath in order to decrease the risk of aliasing.

Consider a sequence of Pk_N control steps over which the nominal schedule is repeated P times; allocation of functional units in the nominal data path is given. An extended set of functional units, consisting of the nominal ones plus those inserted to satisfy Rules 1 and 2 above, if any, is taken into account from now on.

In any control step c_h ($1 \leq h \leq P \cdot k_N - k_{CHECKING}$) and for each ready operation $o_j^C(k)$ of type t_k:

1. Availability of functional units $u_j(k)$ of type t_k compatible with operation $o_j^C(k)$ is determined by examining schedule and allocation of the current replica of the nominal DFG in the step c_h;
2. Allocation of operation $o_j^N(k)$ in the first (*checked*) replica of the nominal DFG is identified to avoid allocating $o_j^C(k)$ on the same functional unit supporting $o_j^N(k)$. (Thus, the same data will never be input to the same unit in both data paths);

$o_j^C(k)$ is then scheduled in step c_h iff a functional unit satisfying both conditions above is available.

For all operations that cannot be scheduled in control step c_h, priorities are updated and the attempt is repeated in the following control step. This mixed scheduling-and-allocation step is repeated until either the whole checking DFG is scheduled in a satisfactory way over the $P \cdot k_N - k_{CHECKING}$ 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: This reduces the risk of aliasing. Moreover, exclusion of register sharing is suggested by the following considerations. First, lifetimes for variables in the checking data path are usually rather long. Second, registers in the nominal data path are already shared as much as possible to reduce the circuit complexity of the nominal architecture. Third, registers in the nominal data path are used P times during operation in the checking data path.

Finally, the interconnection network is created and the control FSM is synthesized.

If no acceptable schedule is found, we increase the set of resources 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:

1. For each type t_k, the number of times that the schedule of ready operations compatible with t_k has been delayed in the checking DFG schedule are counted and the first control step in which such a miss occurred is recorded,
2. A functional unit of the type associated with the highest count and—for equal count value—with the earliest miss is added to the set; in the case of multiple units with identical associated values, 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 algorithm as well as

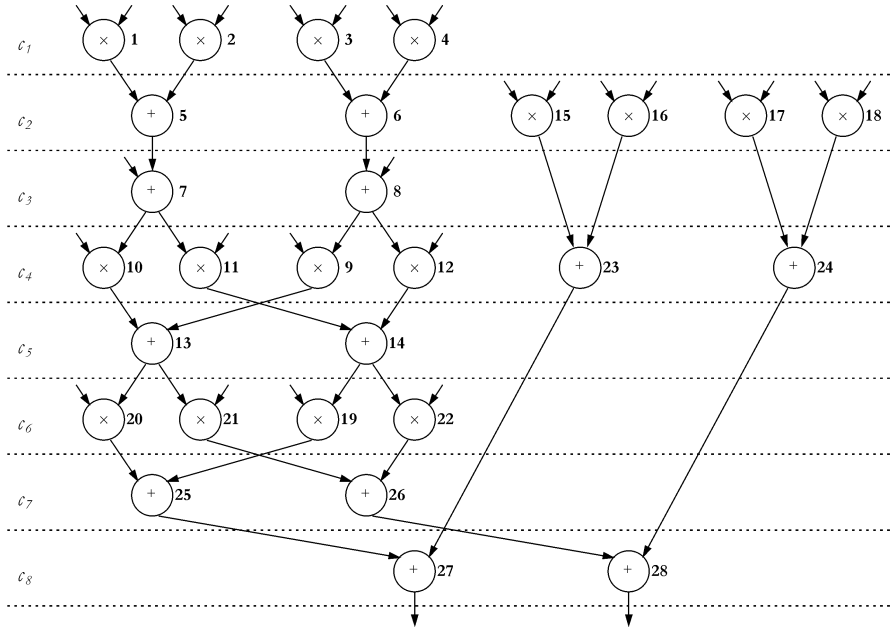

Fig. 1. Optimum scheduling for AR filter.

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 $o_j(k)$ of type t_k in the DFG as the difference $m_j(k)$ between its ALAP and its ASAP labels [14], priority increasing with decreasing mobility values. We suggest here to use *extended mobility* defined as $e_j(k) = (P-1)k_N + m_j(k) - k_{CHECKING}$, 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 it is impossible to schedule, in the present control step, an operation $o_j^C(k)$ such that $e_j^C(k) = 0$, 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 the analysis of the whole DFG.

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

1. At any given control step c_h, for each type t_k of functional unit determine the set $\mathbf{o}_{h,k}^C$ of the ready operations $o_j^C(k)$ that can be allocated on units of this type and the set $\mathbf{U}_{h,k}$ of functional units $u_j(k)$ of type t_k that are not used in the nominal DFG during step c_h.
2. Whenever neither $\mathbf{o}_{h,k}^C$ nor $\mathbf{U}_{h,k}$ are empty, create a bipartite graph whose nodes represent, respectively, elements in $\mathbf{o}_{h,k}^C$ and in $\mathbf{U}_{h,k}$ and a node representing $o_j^C(k) \in \mathbf{o}_{h,k}^C$ is connected by an edge to a node representing $u_j(k) \in \mathbf{U}_{h,k}$ iff operation $o_j^C(k)$ in the nominal DFG has not been allocated to $u_j(k)$. Each edge is labeled with a weight given by the operation's extended mobility $e_j^C(k)$ evaluated at control step c_h.
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 control step c_h and the allocation is given by the matching. Otherwise, an "optimum" partial 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 this last case, failure is declared and a renewed attempt is made with suitably increased set of functional units, following the criterion already specified.

It may be worthwhile noting that the number of control steps finally required by the checking DFG may be less than P_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.

With reference to aliasing, step 2 above 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 Section 5.

As an example, we consider the AR filter also discussed in [4], [15], whose optimum time-constrained schedule—eight control steps long—is given in Fig. 1. 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 possible binding shown in Table 1. It is $k_N = 8$. We assume as periodicity of checking $P = 3$, with

TABLE 1
Allocation and Binding for the Nominal AR Filter

Control step	m1	m2	m3	m4	a1	a2
cs1	O_1	O_2	O_3	O_4		
cs2	O_{15}	O_{16}	O_{17}	O_{18}	O_5	O_6
cs3					O_7	O_8
cs4	O_9	O_{10}	O_{11}	O_{12}	O_{23}	O_{24}
cs5					O_{13}	O_{14}
cs6	O_{19}	O_{20}	O_{21}	O_{22}		
cs7					O_{25}	O_{26}
cs8					O_{27}	O_{28}

no restrictions on the number of checkers (a checking operation requires one control step). Semiconcurrent checking then requires at most *23* control steps to schedule the checking DFG, the *24*th step being reserved for checking. The checking architecture of self-checking reference structure can be scheduled over *21* control steps by using just *one* multiplier, *one* adder, and *one* checker.

Our shared-resources algorithm is now applied, attempting first to schedule a checking DFG within *23* control steps, without increasing the resources. The attempt fails at step *23*, where operations *27* and *28* of the checking DFG 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 summarized in Fig. 2 and Table 2.

Semiconcurrent checking with frequency $P = 2$ (actually higher than initially required) is achieved by introducing just *one* extra adder and *one* checker; results of the checking DFG being available at the *14*th control step, one primary output is checked in the *15*th step and the other one in the *16*th. Cost and performance are thus better than in the reference architecture.

Another example from the high-level synthesis benchmarks [16] is the elliptic filter, whose nominal

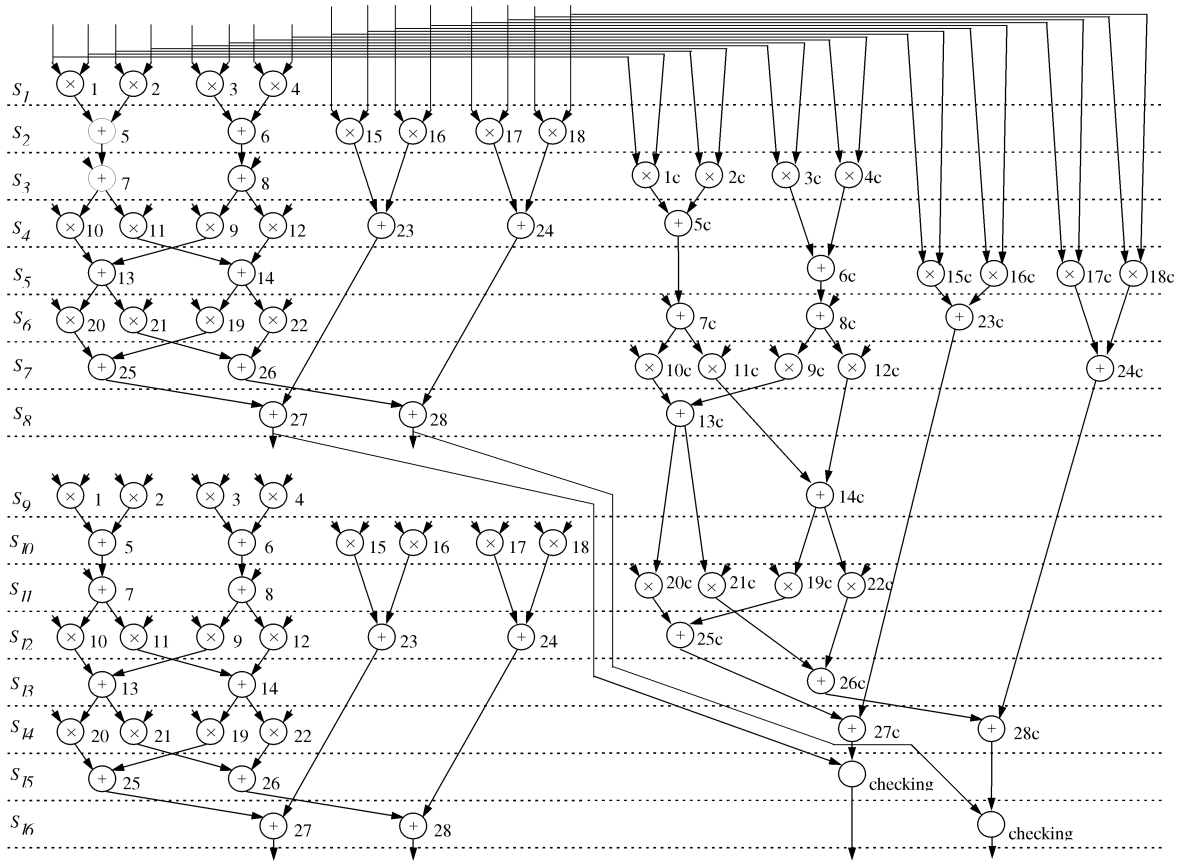

Fig. 2. Scheduling for AR filter with semiconcurrent self-checking capability.

TABLE 2
Allocation and Binding for the Self-Checking AR Filter

Control step	m1	m2	m3	m4	a1	a2	*a3*	*checker*
cs1	O_1	O_2	O_3	O_4				
cs2	O_{15}	O_{16}	O_{17}	O_{18}	O_5	O_6		
cs3	o_2^C	o_1^C	o_4^C	o_3^C	O_7	O_8		
cs4	O_9	O_{10}	O_{11}	O_{12}	O_{23}	O_{24}	o_5^C	
cs5	o_{16}^C	o_{15}^C	o_{18}^C	o_{17}^C	O_{13}	O_{14}	o_6^C	
cs6	O_{19}	O_{20}	O_{21}	O_{22}	o_8^C	o_7^C	o_{23}^C	
cs7	o_{10}^C	o_9^C	o_{12}^C	o_{11}^C	O_{25}	O_{26}	o_{24}^C	
cs8					O_{27}	O_{28}	o_{13}^C	
cs9	O_1	O_2	O_3	O_4			o_{14}^C	
cs10	O_{15}	O_{16}	O_{17}	O_{18}	O_5	O_6		
cs11	o_{20}^C	o_{19}^C	o_{22}^C	o_{21}^C	O_7	O_8		
cs12	O_9	O_{10}	O_{11}	O_{12}	O_{23}	O_{24}	o_{25}^C	
cs13					O_{13}	O_{14}	o_{26}^C	
cs14	O_{19}	O_{20}	O_{21}	O_{22}		o_{27}^C	o_{28}^C	
cs15					O_{25}	O_{26}		*out.* o_{27}
cs16					O_{27}	O_{28}		*out.* o_{28}

(time-constrained) structure requires *four* adders and operates on *11* control steps. The minimum-resource reference architecture involves *one* adder and *one* checker and requires *27* control steps (*26* for the computation and *one* for checking the last-produced output); thus, checking periodicity of *three* is obtained. A shared-resource solution can be designed *without introducing any new adder*; the checking data path operates in *16* control steps so that semiconcurrent checking can be achieved with a periodicity of *two*.

A final word concerns complexity of the controlling FSM. Comparing costs, again, with the reference architecture, it is easily seen that the number of states—therefore, of flip-flops—does not increase so that this cost component is not higher for the shared-resource solution than for the reference architecture. Total cost of the FSM depends on complexity of the control signals forwarded to the data path, thence also on register and interconnection network allocation.

4 Extension of the Approach to Hierarchical Sequencing Graphs

We extend now the basic approach to hierarchical Sequencing Graphs (SGs), a generalization of the DFGs [11]; more precisely, we consider the ones including branch and loop constructs. We do not aim to check the whole SG (i.e., both data and control paths), but only the linear-code DFGs composing the SG. In particular, we show how to apply the basic proposed approach to deal efficiently with these DFGs in branches and loops.

Examine first the case of branching clauses: An elementary example is shown in Fig. 3, the SG consisting of a *head* $\mathrm{DFG_H}$, a *two-way branch* ($\mathrm{DFG_A}$ and $\mathrm{DFG_B}$) and a *tail* $\mathrm{DFG_T}$. (Extension to the case of multiple branchings with simultaneous execution of parallel subgraphs has no conceptual difficulties).

Let E_α^N be the αth execution of the *nominal* SG and let it be a *checked* execution; let E_α^C be the associated *checking* execution. The next checked execution of the SG is $E_{\alpha+P}^N$. The head $\mathrm{DFG_H}$ of E_α^N generates an output that will be used by the controlling FSM to select the execution of the appropriate branch; such output is equivalent to any primary output and, therefore, has to be checked as in SG. Denote by $\mathrm{N_H}$ the number of control steps of the nominal schedule of $\mathrm{DFG_H}$.

Denote by $\mathrm{DFG_{AN}}$ and $\mathrm{DFG_{BN}}$ the two branches in the checked execution of the nominal SG, by $\mathrm{DFG_{AC}}$ and $\mathrm{DFG_{BC}}$ the corresponding branches in the checking architecture, and by $\mathrm{N_A}$ and $\mathrm{N_B}$ the number of control steps of the nominal schedule of the branches. For generality's sake, no assumption is done on the distribution of the actual of the primary inputs, i.e., on the probability of executing each branch. While, obviously, the same branching alternative is chosen for both E_α^N and E_α^C, no such assumption is possible for executions E_β^N, $\alpha < \beta < \alpha + P$, with respect to E_α^C.

In the literature, some algorithms have been proposed to synthesize circuits with optimum of minimum, maximum, or average number of control steps in the critical path. Since we aim to show how semiconcurrent checking can be generally introduced in high-level synthesis, we do not restrict our attention only to these very efficient algorithms.[2]

2. If complex strategies are envisioned to optimize the nominal SG's critical path, the adoption of our approach needs to take into account and use only the hardware resources that have actually been left available by the specific synthesis technique adopted for the nominal architecture.

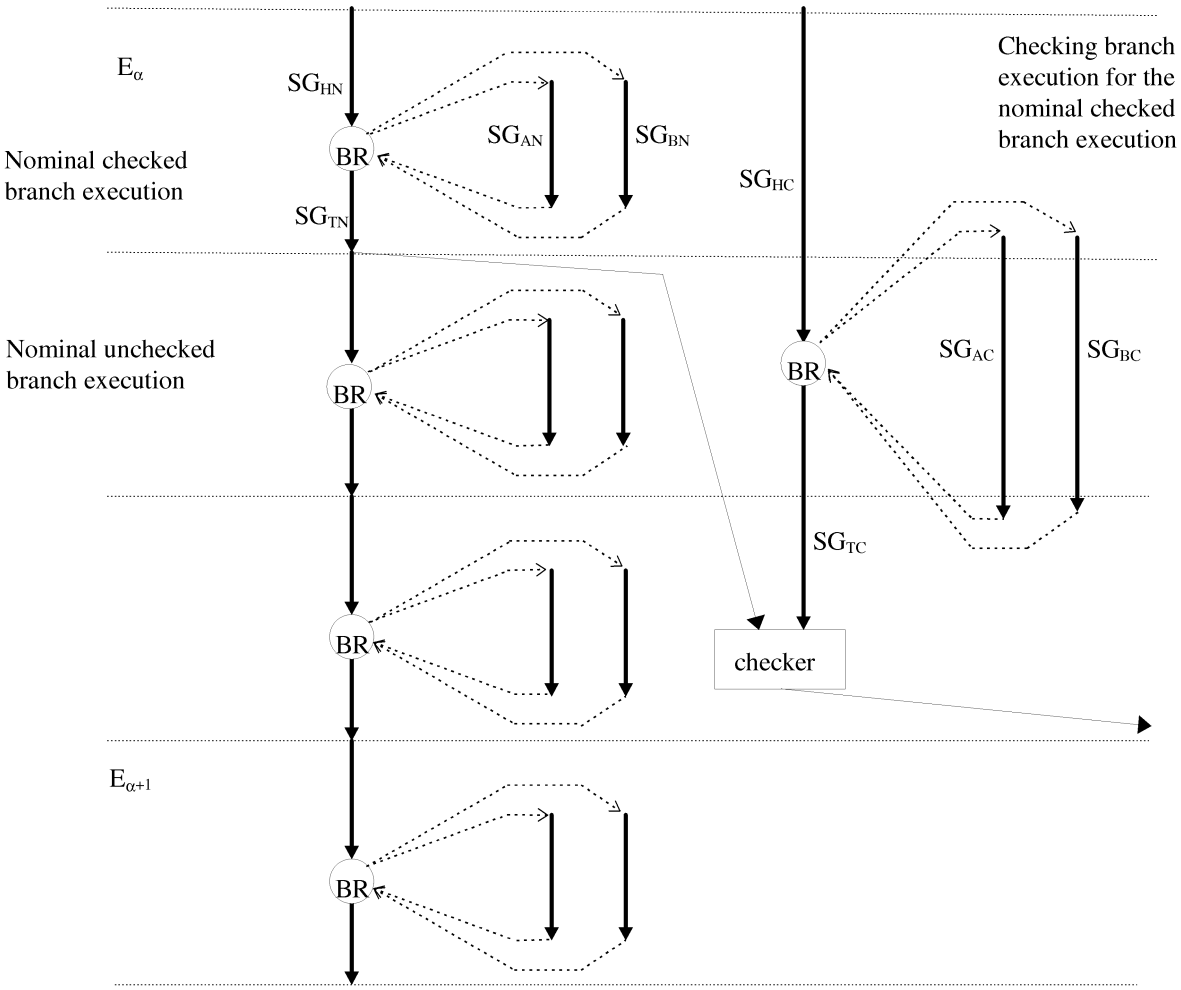

Fig. 3. A nominal and checking data paths with branches.

In particular, we consider here the simple case in which both branches are executed in the same number of control steps so that $\mathrm{DFG_T}$ begins at the same control step, independently from the actually executed branch. On the other hand, this case occurs, for example, whenever the data entering the circuit arrive at a fixed frequency, as in many dedicated signal and image processing applications, and there is no advantage in having different execution time of the branches since the execution frequency must accommodate anyway the worst branch case.

This assumption allows for simpler identification of the functional units that are not used in each control step of the P nominal executions of the branch construct in order to schedule and allocate the operations onto the available functional units while avoiding aliasing. If different branch duration were envisioned, we should consider all possible combinations of branch activation during each execution E^N_β, $\alpha < \beta < \alpha + P$. Identification of the unused functional units would be made difficult by the variable step in the schedule from which the executions begin. Besides, too many opportunities for resource sharing would be discarded to avoid aliasing. On the other hand, this equal duration of branches allows for limiting the complexity of the controlling FSM. More general scheduling and allocation algorithms can be envisioned to achieve higher performance in the case of branches with different duration: The proposed approach can obviously be extended to deal with these solutions.

The tail $\mathrm{DFG_T}$ is the final sequence of operations that is shared between the two branches and concludes the branch construct. It can be void. Denote by $\mathrm{N_T}$ the number of control steps of the nominal schedule of $\mathrm{DFG_T}$.

Under the above assumption, $k_N = \mathrm{N_H} + max(\mathrm{N_A}, \mathrm{N_B}) + \mathrm{N_T}$ is the number of control steps of the nominal SG. Let $\mathrm{C_H}$, $\mathrm{C_A}$, $\mathrm{C_B}$, $\mathrm{C_T}$, and k_C be the corresponding figures in the checking architecture. Let c_h be the control step reached in design of the checking data path.

The basic algorithm presented in Section 3 is modified to guarantee availability of resources independently of the alternative chosen in each iteration of the nominal and the checking executions; modifications concern the rules for constructing the bipartite graphs. More specifically:

- For any step c_h, if scheduled operations of the *nominal* SG and ready (but unscheduled) operations of the *checking* SG are within $\mathrm{DFG_H}$ or $\mathrm{DFG_T}$, the basic algorithm is applied.
- For any step c_h, if the scheduled operations of the nominal SG belong to $\mathrm{DFG_H}$ or $\mathrm{DFG_T}$, while the ready (and unscheduled) operations of the checking

SG belong to $\mathrm{DFG_A}$ or $\mathrm{DFG_B}$, a bipartite graph is built as in the basic algorithm separately for each of the branches in the checking SG. The basic algorithm independently performs scheduling and allocation for each branch.

- For any step c_h, if the scheduled operations of the nominal SG belong to $\mathrm{DFG_A}$ or $\mathrm{DFG_B}$, but the ones of the checking SG belong to $\mathrm{DFG_H}$ or $\mathrm{DFG_T}$, the set of available functional units $\mathbf{U}_{h,k}$ is obtained by removing the functional units $u_j(k)$ 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 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 Step 1 of the basic algorithm to evaluate aliasing correctly:

  1. one bipartite graph including operations of branch $\mathrm{DFG_{AC}}$ and functional units $u_j(k)$ not used by branch $\mathrm{DFG_{AN}}$ in the same step c_h,
  2. a corresponding one for branches $\mathrm{DFG_{BC}}$ $\mathrm{DFG_{BN}}$.

  Steps 2 and 3 of the basic algorithm are 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, again to evaluate aliasing correctly, one bipartite graph is built for each branch of the checking SG as follows, by extending Step 1 of the basic algorithm:

  1. the set of available functional units in either graph ($\mathbf{U}^{\mathbf{A}}_{h,k}$ and $\mathbf{U}^{\mathbf{B}}_{j,k}$) is obtained by removing the functional units $u_j(k)$ allocated in *any* branch of the nominal SG,
  2. operations $o^C_j(k)$ 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 in Step 2 intended to avoid aliasing always refer to the operations in E^N_α.

To show clearly the application of this technique, we consider the SG in Fig. 4a, having empty head and tail sections. Scheduling of the nominal SG by a time constrained approach is shown in the leftmost part of Fig. 4b; 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 three control steps to be completed; allocation of the nominal SG can be performed as shown in Table 3 by using *two* multipliers, *two* adders, and *one* subtractor.

The checking SG built according to our modified algorithm is shown in the rightmost part of Fig. 4b, while the resulting allocation leading to low aliasing probability is given in Table 4. The scheduled checking SG requires *one* additional subtractor (to avoid aliasing) and *one* checker; generation of the checking result is performed in *five* control steps, while the checker can operate in the *six*th control step, allowing a relative checking periodicity P equal to *two*.

A second extension concern *loops*. Consider first the case of *data-independent* numbers of iterations. We discuss here the case of a single loop; extension to multiple and/or nested loops does not require further theoretical analysis. Refer to Fig. 5; notation is similar to that in Fig. 3, $\mathrm{DFG_B}$ here denoting the *loop body*. The nominal schedule requires $\mathrm{N_H}$ control steps for the head, $\mathrm{N_B}$ for a single iteration of the body, and $\mathrm{N_T}$ 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 $\mathrm{C_H}$, $\mathrm{C_B}$, and $\mathrm{C_T}$ The total latency k_N (in number of steps) for the nominal SG is $k_N = \mathrm{N_H} + K\mathrm{N_B} + \mathrm{N_T}$; for the checking SG, $k_C = \mathrm{C_H} + K\mathrm{C_B} + \mathrm{C_T}$. As for linear DFGs, we assume that the optimum schedule-and-allocation choice for the nominal SG is kept unchanged when scheduling and allocation are performed for the checking SG.

A trivial solution consists of complete unrolling of both nominal and checking SGs (possible consequent algorithmic optimizations, if any, must be identical for both): The basic algorithm is thus applied in a straightforward way. While possibly leading to lower latency, this requires a larger number of *control words* for the controlling FSM.

Let us examine an alternative approach without unrolling. Assume that control step c_h has been reached during scheduling and allocation of the checking SG. c_h corresponds to the control step $c^*_h = |c_h - 1|_{\mathrm{mod}\ k_N} + 1$ in the nominal SG. It is:

- If $1 \leq c^*_h \leq \mathrm{N_H}$, 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 $\mathrm{N_H} + 1 \leq c^*_h \leq \mathrm{N_H} + K\mathrm{N_B}$, the step reached within execution of the cycle in the nominal SG is $|c_h - N_H - 1|_{\mathrm{mod}\ N_B} + 1$; this information is used for the checking data path, as above;
- If $\mathrm{N_H} + K\mathrm{N_B} + 1 \leq c^*_h \leq k_N$, the basic algorithm is again applied.

Checking in this way is implicitly applied to all operations executed during all K iterations of the loop. Since checking periodicity is immediately related to $k_C - k_N$, for large values of K and of $\mathrm{C_B} - \mathrm{C_H}$, it may well happen that—in order to achieve acceptable checking periodicity—the set of resources to be introduced for implementation of the checking SG will become too relevant.

A solution that limits the increase of resources may be defined at a lower level by checking each individual section of the loop only once within the defined periodicity. Checking is applied independently to the head section, to the last iteration of the loop body, and to the tail section of the SG. The checking architecture is designed to operate on the body section as follows:

1. At the beginning of a checked execution, input data are fed to both nominal and checking data paths. Execution of the head section continues in both data paths as if such sections were independent SGs so that "results" are checked as soon as available;

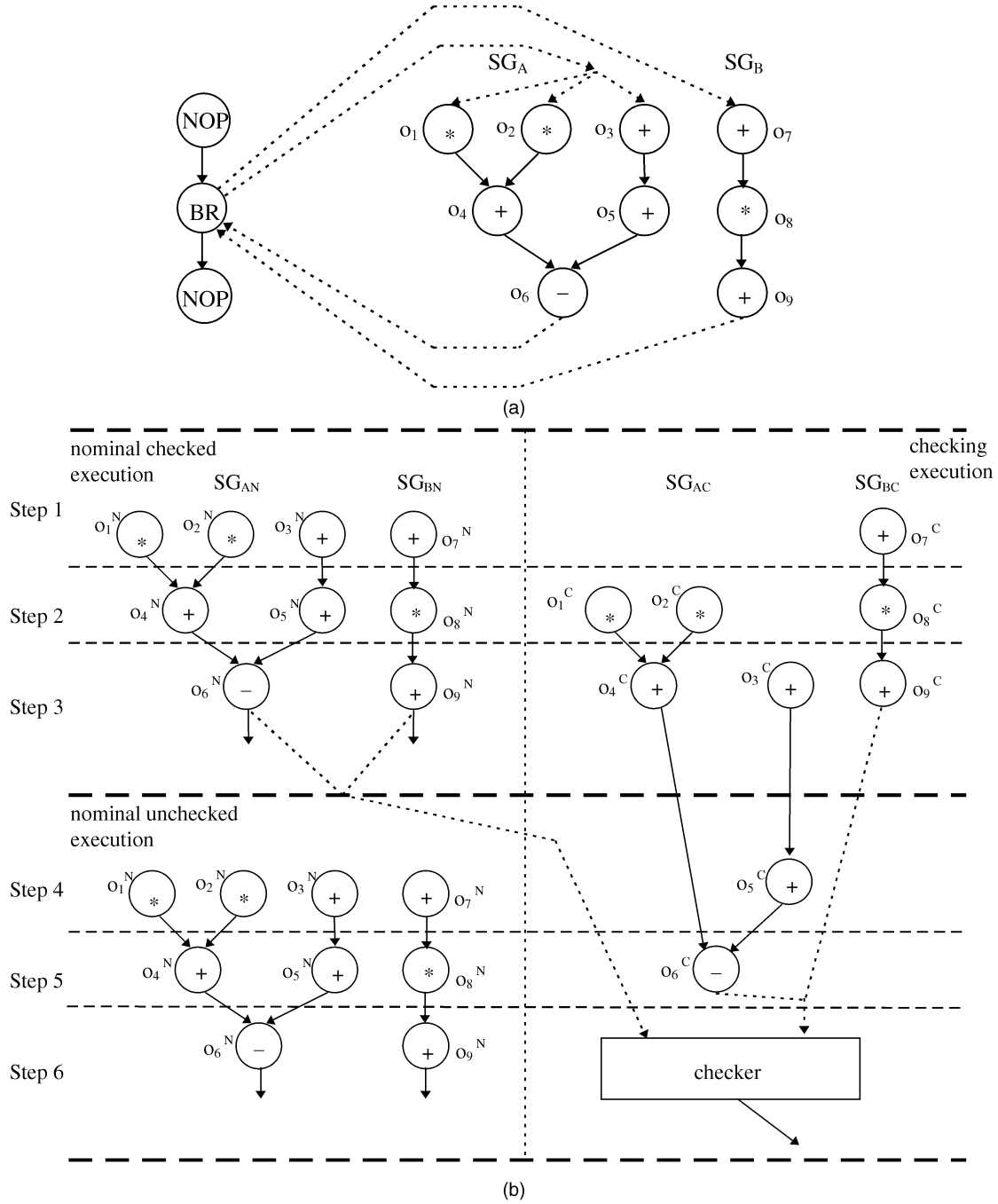

Fig. 4. Application of the modified algorithm for data paths with branches: (a) the SG, (b) the scheduled nominal and checking SGs.

2. At step N_H, the nominal architecture stores data input to the loop body as checked loop body inputs.

TABLE 3
Allocation and Binding for the Nominal SG for the Example of Fig. 4

Control step	m1	m2	a1	a2	s1
cs1	O_1,-	O_2,-	O_3,O_7		
cs2	-,O_8		O_5,-	O_4,-	
cs3			-,O_9		O_6,-

3. Execution of the current iteration of the loop body is performed by the nominal architecture; at some control step within it, the checking architecture may in turn start execution of the loop body on the current checked loop body inputs. At the end of the current iteration by the nominal architecture, updated values of the loop body inputs are stored as *checked* loop body inputs, replacing the previous ones; if the checking architecture is already executing the loop body, intermediate results achieved are discarded and the checking computation is restarted on the new checked body inputs. Step 2 is repeated

TABLE 4
Allocation and Binding for the Checking SG for the Example of Fig. 4

Control step	m1	m2	a1	a2	s1	*s2*	*checker*
cs1	O_1,-	O_2,-	O_3,O_7	-,O_7^C			
cs2	O_2^C,O_8	O_1^C,O_8^C	O_5,-	O_4,-			
cs3			O_4^C,O_9	O_3^C,O_9^C	O_6,-		
cs4	O_1,-	O_2,-	O_3,O_7	O_5^C,-			
cs5	-,O_8		O_5,-	O_4,-		O_6^C,-	
cs6			-,O_9		O_6,-		***out.*** O_6,O_9

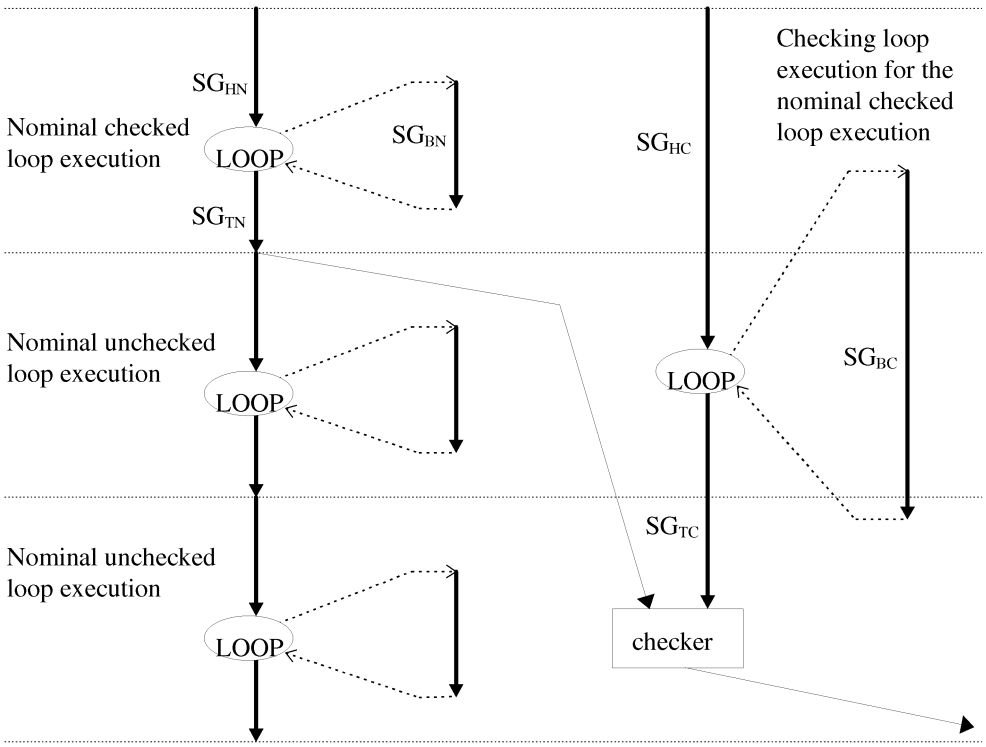

Fig. 5. Nominal and checking data paths with loop.

for each iteration by the nominal architecture excepting the last one.

4. When the nominal architecture starts the Kth iteration of the loop body, the checking architecture is allowed to complete the computation of the loop body on the final checked body inputs. At this point, results of the nominal and checking final iterations of the loop are compared.
5. When the nominal architecture completes the loop body iterations, all nominal results relevant to the tail section are stored as checked tail inputs. Checking of the tail section is then performed as for any linear-code DFG.

The above solution allows us to reduce latency of execution on the checking architecture with respect to the nominal one; on the other hand, it also reduces fault coverage since it reduces probability of fault excitation (the loop body is tested with one instead of K vectors).

To be consistent with evaluations produced for linear-code DFGs, the reference architecture ought to be designed for the same checking philosophy (i.e., with checking involving the last iteration of the loop); therefore, the reference controlling FSM would have at least as many states as for the resource sharing solution.

This last solution can be adopted in the case of *data-dependent* loops as well. Step 3 above is simply activated at the last loop body iteration.[3] While, conceptually, the scheduling-and-allocation technique for the checking SG is not modified, in practice, its application becomes more complex. As an example, we consider the differential equation solver [16]; the resulting self-checking SG is shown in Fig. 6, while allocation is given in Table 5. The nominal SG operates in *seven* control steps and requires *one*

3. On the basis of the definition given in Section 2, periodicity P would become, in this case, data-dependent as well; to make scheduling and allocation feasible, a fixed value of P is required. The simplest solution is to consider nominal executions in which the loop body is executed once only and to refer P to such value.

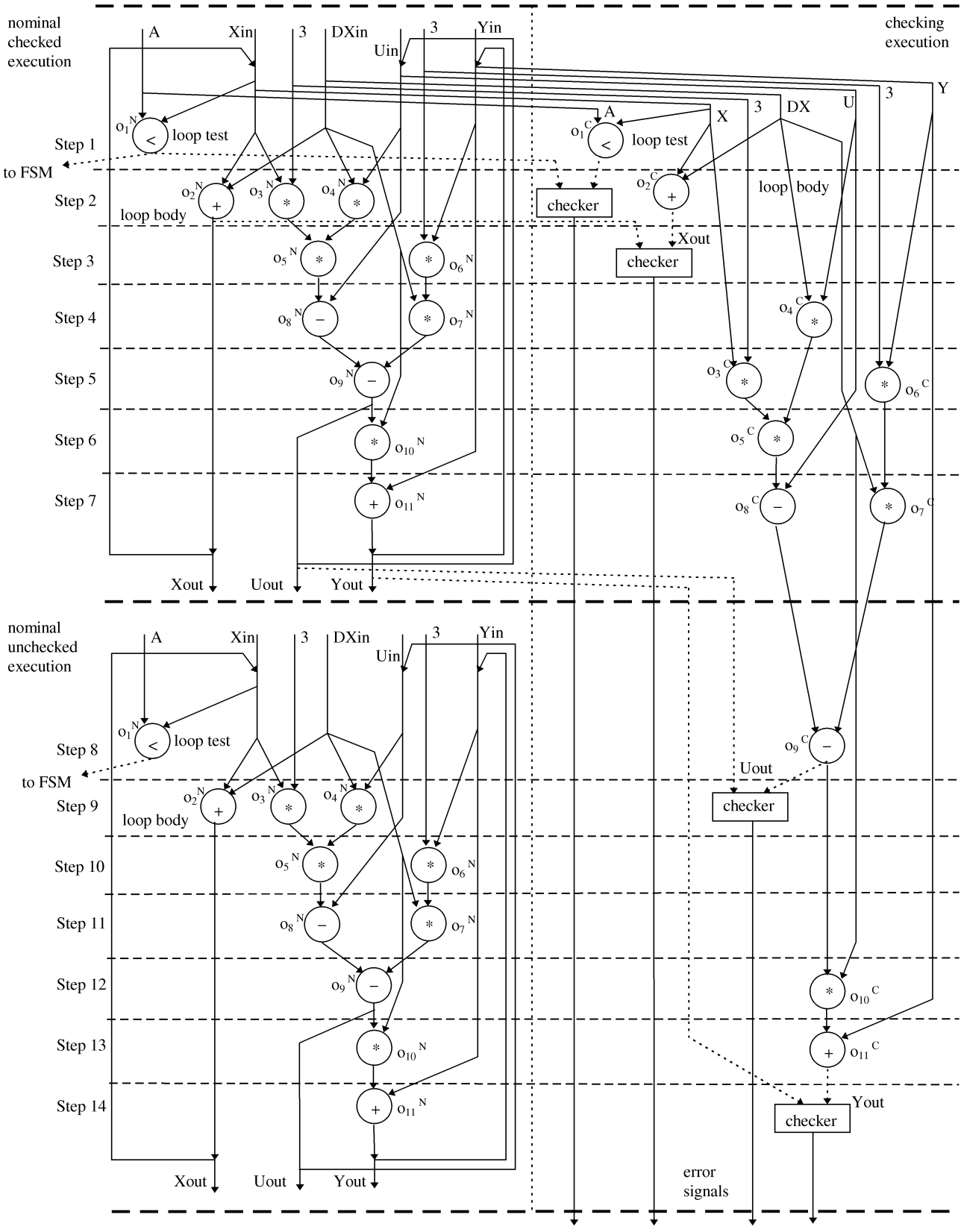

Fig. 6. The differential equation circuit.

comparator, *two* multiplier, *one* adder, and *one* subtractor. The minimum additional functional units needed to avoid aliasing in the checking SG are *one* comparator, *one* adder, and *one* subtractor: Checking can be performed in *10* control steps and, as a consequence, the relative periodicity P can be equal to *two*.

TABLE 5
Allocation and Binding for the Self-Checking SG for the example of Fig. 6

Control step	m1	m2	a1	*a2*	s1	*s2*	c1	*c2*	*checker*
cs1							O_1	o_1^C	
cs2	O_3	O_4	O_2	o_2^C					***out. o1***
cs3	O_5	O_6							***out. o2***
cs4	o_4^C	O_7			O_8				
cs5	o_6^C	o_3^C			O_9				
cs6	O_{10}	o_5^C							
cs7	o_7^C		O_{11}			o_8^C			
cs8						o_9^C	O_1		
cs9	O_3	O_4	O_2						***out. o9***
cs10	O_5	O_6							
cs11		O_7			O_8				
cs12		o_{10}^C			O_9				
cs13	O_{10}			o_{11}^C					
cs14			O_{11}						***out. o11***

If nested loops—possibly enclosing branches—are considered, the complexity of both the algorithm and the controlling FSM may become so relevant as to make straightforward duplication with comparison and concurrent checking attractive. (It may also be recalled that, when SGs with complex control structures are considered, some authors suggest control-flow oriented scheduling techniques [17], [18] rather than the data-flow oriented ones adopted here).

5 The Aliasing Problem

As suggested in Section 2, resource sharing between nominal and checking data paths may lead to risk of aliasing; Condition 1 of Section 3 guiding allocation of the checking DFG is only the minimal condition to be satisfied to avoid the *certainty* of aliasing on the individual operation.

Let's consider first the case of sequences of operations, with specific reference to linear DFGs. 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 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 the 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 $o_1(o_2(\alpha, \beta, \gamma))$: Assume the presence of two identical functional units u_1, u_2 whose allocation leads, in the nominal DFG, to $o_1 \Rightarrow u_1, o_2 \Rightarrow u_2$, and, in the checking DFG, to $o_1 \Rightarrow u_2, o_2 \Rightarrow u_1$. Assume further that u_1 is faulty and u_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 $f_1^*(f_2(\alpha, \beta, \gamma))$ and as $f_2(f_1^*(\alpha, \beta), \gamma))$. Aliasing occurs only if $f_1^*(f_2(\alpha, \beta)\gamma)) = f_2(f_1^*(\alpha, \beta)\gamma))$ and if results are affected by an error. Possibilities are as follows:

- $f_2(\alpha, \beta) = f_1^*(\alpha, \beta)$: This occurs only if the inputs α and β do not excite the fault. In this case, results are both error-free (error *masking*, not *aliasing*, occurs). Then:
  - If the pair of inputs $f_2(\alpha, \beta), \gamma$ excites the fault in u_1, results produced by the two sequences will be different and no aliasing on the whole sequence is possible,
  - Otherwise, results of sequence $f_1^*(f_2(\alpha, \beta)\gamma))$ are affected by error and different from the ones of sequence $f_2(f_1^*(\alpha, \beta)\gamma))$: No aliasing occurs;
- $f_2(\alpha, \beta) \neq f_1^*(\alpha, \beta)$: inputs α, β excite the fault in fu_1. Denote $f_2(\alpha, \beta) = \delta, f_1^*(\alpha, \beta) = \delta^*$. Then:
  - If inputs δ, γ do not excite the fault, it is $f_2(\delta, \gamma) = f_1^*(\delta, \gamma)$; if $f_2(\delta, \gamma) \neq f_2(\delta^*, \gamma)$, the error is detected; otherwise, both results are correct and error *masking* occurs;
  - If inputs δ, γ excite the fault, it is $f_2(\delta, \gamma) \neq f_1^*(\delta, \gamma)$; in this case, *aliasing* may occur if $f_1^*(\delta, \gamma) = f_2(\delta^*, \gamma)$.

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. It is 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.

As already said in Section 2, in aliasing analysis, faults in registers and interconnection network can be viewed as faults in the functional units computing the values stored in the registers.

Let's consider now the case of hierarchical SGs, in particular, the ones containing branches. Since Step 2 of the scheduling and allocation algorithm discussed in Section 4 is performed as in the basic algorithm discussed in Section 3, the same conditions concerning aliasing for individual operations and sequences of operations hold in SG with branches as well.

To reduce the probability of excessive error latency due to lack of fault excitation in the faulty functional units, a further design guideline can be adopted. Consider the case in which some functional units can be used in both branches $\mathrm{DFG_A}$ and $\mathrm{DFG_B}$ at control step c_h (i.e., that belong both to $\mathbf{U}^{\mathbf{A}}_{h,k}$ and $\mathbf{U}^{\mathbf{B}}_{h,k}$) as well as at control step c_{h+a} (i.e., that belong both to $\mathbf{U}^{\mathbf{A}}_{h+a,k}$ and $\mathbf{U}^{\mathbf{B}}_{h+a,k}$). The units among them that have been allocated to operations in branch $\mathrm{DFG_A}$ at control step c_h should be used for operations in branch $\mathrm{DFG_B}$ at control step c_{h+a} and vice versa to guarantee uniform probability of usage to all units and, as a consequence, uniform probability of fault excitation.

As an example, consider the dataflow graph describing $x + y + z$. The reference architecture consists of the adder u_1. The first and the second additions are performed by u_1 at the control steps c_1 and c_2, respectively. The use of the same unit to perform checking in subsequent steps will lead to aliasing. In fact, an error ε_1 will be added to the nominal addition $x + y$ and the error ε_2 will be then added to the subsequent addition in which z is added. In the checking computation, the same errors will be added to the addition. The use of a second functional unit u_2 to perform checking avoids this effect under the single fault assumption, thus allowing error detection. Consider the mapping of the addition $x + y$ of the nominal computation on the unit u_1 and that of the addition $x + y$ of the checking computation on the unit u_2. Let's map the addition of z in the nominal computation on the unit u_2 and that in the checking computation on unit u_1. A fault in u_1 may appear as the error ε_1 in the first addition of the nominal computation *and* in the second addition of the checking computation, thus resulting in aliasing since the same error ε_1 is added to $x + y + z$, both in the nominal and in the checking data path.

To reduce the probability of aliasing, suitable techniques can also be used to synthesize the data path. An interesting example is provided in [19]. The strategy proposed there can be also included in high-level synthesis with semiconcurrent checking so as to further reduce the need for redundant hardware resources. We have not based this paper on such a technique in order to show the generality of our strategy without binding it to a specific scheduling and allocation algorithm.

6 Experimental Evaluation

To evaluate the effectiveness of the proposed strategy to achieve semiconcurrent error detection at a reasonable cost in terms of circuit complexity, we synthesized some typical circuits available in the literature. As significant and characteristic examples we considered the AR filter, the elliptical filter, the differential equation solver, and the branch circuit reported in Fig. 4. Specifically, we synthesized the nominal data path without error detection ability, the modular redundancy approach based on duplication with output comparison, the reference architecture described in Section 2, and the self-checking architecture obtained by using the innovative synthesis strategy proposed in this paper. Since the circuit complexity saving of our strategy depends on the relative checking periodicity, P, the analysis was performed for different values of this parameter that characterizes the frequency of the detection operation. Synthesis has been performed by generating the corresponding VHDL description of the circuits and by using Synopsys Behavioral Compiler v. 2000.5 and Synopsys Design Compiler v. 2000.5.

In Tables 6, 7, and 8, we report the results of our experiments. In Table 6, for each circuit, we give the number of adders, multipliers, subtractors, comparators, and registers that are required in the self-checking strategies mentioned above; for simplicity's sake, the evaluation is given with $P = 2$ only for the AR filter. In Table 7, we summarize the overall circuit complexity of the envisioned self-checking circuits; the circuit complexity is given in equivalent gate count, as obtained by Synopsys Behavioral Compiler v. 2000.5 and Synopsys Design Compiler v. 2000.5. In Table 8, we give the circuit complexity reduction (in percent) of the semiconcurrent approaches with respect to the duplication with comparison. Results concerning the reference architecture and the proposed self-checking strategy are shown to distinguish the contribution to circuit complexity reduction generally given by the semiconcurrent checking from the specific contribution given by our strategy.

The goal of our analysis given in Table 8 is two-fold. First, we show that our self-checking strategy is effective to save circuit complexity with respect to the conventional duplication with comparison. This is achieved at the expense of a reduction in the fail safety since only one out of P results are checked. The less frequent the checking is, the higher the circuit complexity reduction that can be achieved is. Aliasing limits the circuit saving at higher values of P since it makes it mandatory to limit the resource reuse and introduce new resources. Second, we show that our strategy is also able to save circuit complexity with respect to the reference architecture presented in Section 2. In some cases, our strategy largely outperforms the basic reference architecture approach. In general, the amount of circuit complexity saving provided by the proposed self-checking strategy depends on the structure of the computation, i.e., on the aliasing induced by the specific resource reuse. It is worth noting that this limit is not due to the proposed strategy, but to the intrinsic nature of the circuit structure.

TABLE 6
Circuit Complexity in Terms of Functional Units and Registers

Circuit	# adders	# multipliers	# subtractors	# disequality (<) comparators	# equality comparators	# registers
AR filter – nominal architecture	2	4				6
AR filter – modular redundancy architecture	4	8			2	12
AR filter – reference architecture *P=2*	3	6			1	14
AR filter – self-checking architecture *P=2*	3	4			1	11
AR filter – reference architecture *P=4*	3	5			1	15
AR filter – self-checking architecture *P=4*	2	4			1	12
Elliptical filter – nominal architecture	3					6
Elliptical filter – modular redundancy architecture	6				8	12
Elliptical filter – reference architecture *P=2*	4				7	21
Elliptical filter – self-checking architecture *P=2*	4				7	12
Differential equation – nominal architecture	1	2	1	1		7
Differential equation – modular redundancy	2	4	2	2	3	14
Differential equation – reference architecture *P=2*	2	4	2	2	2	17
Differential equation – self-checking architecture *P=2*	2	2	2	2	1	10
Branch – nominal architecture	2	2	1			3
Branch – modular redundancy architecture	4	4	2		1	6
Branch – reference architecture *P=2*	3	3	2		1	7
Branch self-checking architecture *P=2*	2	2	2		1	4

TABLE 7
Circuit Complexity in Equivalent Gates

Circuit	Nominal architecture	Modular redundancy architecture	Reference architecture *P=2*	Self-checking architecture *P=2*	Reference architecture *P=4*	Self-checking architecture *P=4*
AR filter	7721	15559	13750	10629	12728	10747
Elliptical filter	3433	7335	9988	6889	9946	6976
Differential equation	5165	10511	10362	6532	10494	6674
Branch	3863	7784	6945	5002	8011	5040

TABLE 8
Circuit Complexity Reduction (Percentage) with Respect to the Modular Redundancy Architecture (the Circuit Complexity Is Measured in Equivalent Gates)

Circuit	Reference architecture *P=2*	Self-checking architecture *P=2*	Reference architecture *P=4*	Self-checking architecture *P=4*
AR filter	11,63%	31,69%	18,20%	30,93%
Elliptical filter	-36,17%	6,08%	-35,60%	4,89%
Differential equation	1,42%	37,86%	0,16%	36,50%
Branch	10,78%	35,74%	-2,92%	35,25%

From Table 8, the designer can choose the most suitable value of P as a trade-off between the need for limiting the possible propagation of erroneous results and the circuit complexity.

As far as circuit latency is concerned, it can be minimized in all solutions by adopting a suited design approach. In duplication with comparison, checking needs further steps after the computation of the results. However, we can

accommodate the result checking in the first step of the subsequent iteration so that latency will not be increased; correctness of each final result will be asserted one step after result delivery. In the reference architecture, as well as in our strategy, we perform schedule and allocation so that checking is always performed without any latency increase within the P iterations.

The design strategy proposed here constitutes the basis for an experimental tool that has been implemented in the C++ language on an IBM-compatible PC, running the MS-Windows/NT operating system. The designer can define the SG by interacting with a graphic interface. By means of visual interaction, the designer is then guided through the analysis and synthesis of self-checking solutions having different checking periodicity and redundancy in order to allow her/him to identify the structure that best matches the application requirements. The output of our experimental tool is the VHDL description of the self-checking data path and the schedule to be implemented in the controlling FSM: This compatibility and integrability with commercial CAD tools will allow for including our approach in a standard design flow and CAD environment.

7 Conclusions

In this paper, we have shown how to effectively use a semiconcurrent approach in high-level synthesis environments to design self-checking circuits. The reference architecture has been identified and constructed by using any scheduling and allocation algorithm. Then, the nominal data path has been extended to include self-checking features. The nominal architecture with the related schedule and allocation of operations onto resources is considered frozen and never changed to achieve the self-checking features. The hardware resources of the nominal data path that are unused at each clock step are exploited in the checking one whenever the detection ability is not impaired by possible aliasing. This minimizes the need for introducing redundant resources to implement the checking data path. The proposed design strategy has been shown viable and effective by means of popular benchmarks.

Although the basic idea can be incorporated in any scheduling and allocation algorithm, as an example, a very simple algorithm has been adopted to create the checking data path. Our focus was not in fact on the optimization of the algorithm: We aimed to show the feasibility and effectiveness of introducing the self-checking properties (namely by a semiconcurrent approach) in the high-level synthesis environment directly.

References

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

[2] J.E. Carletta and C.A. Papachristou, "Behavioral Testability Insertion for Datapath-Controller Circuits," *J. Electronic Testing*, vol. 11, no. 1, pp. 9-28, Aug. 1997.

[3] L. Avra, "Allocation and Assignment in High-Level Synthesis for Self-Testable Data-Paths," *Proc. Int'l Test Conf.*, pp. 463-472, 1991.

[4] A. Orairoglu and R. Karri, "Automatic Synthesis of Self-Recovering VLSI Systems," *IEEE Trans. Computers*, vol. 45, no. 2, pp. 131-142, Feb. 1996.

[5] S.S. Ravi, R. Narasimhan, and D.J. Rosenkrantz, "Efficient Algorithms for Analyzing and Synthesizing Fault-Tolerant Datapaths," *Proc. 1995 IEEE Int'l Workshop Defect and Fault Tolerance in VLSI Systems*, Nov. 1995.

[6] A. Antola, V. Piuri, and M.G. Sami, "Optimising High-Level Synthesis for Self-Checking Arithmetic Circuits," *Proc. 1996 IEEE Int'l Symp. Defect and Fault Tolerance in VLSI Systems*, Nov. 1996.

[7] A. Antola, V. Piuri, and M.G. Sami, "A High-Level Synthesis Approach to Optimum Design of Self-Checking Circuits," *Proc. European Design Automation Conf. (EURODAC '96)*, Sept. 1996.

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

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

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

[11] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

[12] A. Antola, V. Piuri, and M.G. Sami, "Semi-Concurrent Error Detection in Data Paths," *Proc. 1997 IEEE Int'l Symp. Defect and Fault Tolerance in VLSI Systems*, Oct. 1997.

[13] A. Antola, V. Piuri, and M.G. Sami, "A Low-Redundancy Approach to Semi-Concurrent Error Detection in Data Paths," *Proc. Design, Automation, and Test in Europe Conf. (DATE98)*, Feb. 1998.

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

[15] G. Buonanno, M. Pugassi, and M.G. Sami, "A High-Level Synthesis Approach to Design of Fault-Tolerant Systems," *Proc. 1997 IEEE VLSI Test Symp.*, Apr. 1997.

[16] N. Dutt and C. Ramachandran, "Benchmarks for the 1992 High Level Syntehsis Workshop," Technical Report #92-107, Univ. of California at Irvine, Oct. 1992.

[17] R. Camposano, "Path-Based Scheduling for Synthesis," *IEEE Trans. Computer-Aided Design*, vol. 10, no. 1, Jan. 1991.

[18] R.A. Bergamaschi, S. Raje, I. Nair, and L. Trevillyan, "Control-Flow Versus Data-Flow-Based Scheduling: Combining Both Approaches in an Adaptive Scheduling System," *IEEE Trans. Very Large Scale Integration Systems*, vol. 5, no. 1, Mar. 1997.

[19] G. Lakshminarayana, A. Raghunathan, and N.K. Jha, "Behavioral Synthesis of Fault Secure Controller/Datapaths Based on Aliasing Probability Analysis," *IEEE Trans. Computers*, vol. 49, no. 9, Sept. 2000.

[20] D.M. Blough, F.J. Kurdahi, and S.Y. Ohm, "High-Level Synthesis of Recoverable VLSI Microarchitectures," *IEEE Trans. Very Large Scale Integration Systems*, vol. 7, no. 4, Dec. 1999.

Anna Antola received the DrEng degree in electronics engineering from the Politecnico di Milano in 1983 and the PhD degree from the Politecnico di Milano in 1989. From 1989 to 1992, she was a researcher at the C.N.R. (Italian National Research Council). From 1992 to 1995, she has been an associate professor of computer science at the Università di Pavia. Since 1995, she is an associate professor of computer science at the Politecnico di Milano. Her research interests include dedicated architectures for image and signal processing, VLSI and WSI devices, high-level and application specific synthesis, self-checking high-level synthesis techniques, and fault and defect tolerance. Prof. Antola is a member of the IEEE, the IEEE Computer Society, and EUROMICRO.

Fabrizio Ferrandi received the DrEng degree (cum laude) in electronical engineering from the Politecnico di Milano, Italy, in 1992, and the PhD degree in information and automation engineering (computer engineering) from the Politecnico di Milano in 1997. Currently, he is an assistant professor of electrical and computer engineering at the Politecnico di Milano. His research interests include synthesis, verification, simulation, and testing of digital circuits and systems. He is a member of the IEEE and the IEEE Computer Society.

Vincenzo Piuri obtained the PhD degree in computer sngineering in 1989 from the Politecnico di Milano. From 1992 to September 2000, he was an associate professor in operating systems at the Politecnico di Milano. Since October 2000, he has been a full professor in computer engineering at the University of Milano. His research interests include computer arithmetic, application-specific processing architectures, fault tolerance, theory, and industrial applications of neural networks. His results have been published in more than 120 papers in books, international journals, and proceedings of international conferences. He is a senior member of the IEEE and a member of the ACM, IMACS, INNS, and AEI. On the IEEE Test Technology Technical Council, he is chair of the Technical Committee on Defect and Fault Tolerance. He has been an associate editor of the *IEEE Transactions on Instrumentation and Measurement* since 1998 and the *IEEE Transactions on Neural Networks* since 2001.

Mariagiovanna Sami obtained the DrEng degree in electronic engineering from the Politecnico di Milano in 1966 and her Libera Docenza (Computing and Switching Theory) in 1971. Since graduation, she has been with the Department of Electronics and Information, Politecnico di Milano, where she obtained various research and teaching positions and where she has been a full professor since 1980. Her research interests are in the areas of fault tolerance, highly parallel VLSI and WSI architectures, and high-level synthesis. She has published more than 150 papers in international journals and conference proceedings; she is coauthor of the book *Fault Tolerance through Reconfiguration in VLSI and WSI Arrays*, published by MIT Press. She is a senior member of the IEEE and a member of EUROMICRO. She was editor-in-chief of the *EUROMICRO Journal* and she was a member the editorial board of the *IEEE Transactions on Computers*.