Improving Testability of Non-Scan Designs during Behavioral Synthesis

Marie-Lise Flottes, D. Hammad, Bruno Rouzeyre

Improving Testability of Non-Scan Designs during Behavioral Synthesis

M.L. Flottes, D. Hammad, B. Rouzeyre
Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM)
U.M. CNRS 5506, Université de Montpellier II.
161 rue Ada, 34392 Montpellier Cedex 5, FRANCE

Abstract

We present a behavioral synthesis method aimed at generating testable datapaths. A non-scan testing strategy is targeted. Given performance and area constraints, the system is aimed at seeking among potential design alternatives the one presenting the least testability problems. The backbone of this methodology is a testability analysis method that works at different abstraction levels of the design description -from strictly behavioral domain to purely structural domain-. Considering a partially mapped behavioral specification, the testability analysis identifies the testability problems of the future structure. These problems are solved along the synthesis process, for example during the register allocation/binding task as presented in this paper.

I Introduction

High Level Synthesis (HLS) systems generate a structural hardware implementation at Register Transfer Level (RTL), given a behavioral description of design and user defined constraints like area and performance. The behavioral description is first translated into an intermediate description such as a Data Flow Graph (DFG) or a Control Flow Graph (CFG). Nodes and arcs in a DFG correspond respectively to operations and variables in the behavioral description. In a CFG, arcs represent sequencing and nodes define data transformations or control branching. Then, HLS algorithms act on this intermediate description. They perform scheduling (partitioning the description into time intervals called control steps), binding (assigning variables to storage components and operations to functional units), as well as generating interconnect network and control logic. The final output is an RTL structure composed of a datapath and of a controller. The datapath contains the functional units (f.u.s), the storage components, and the interconnect network connecting these various physical resources. The controller is a finite-state machine which drives the control signals of datapath elements in order to achieve the intended behavior.

Progress in technology and CAD tools has allowed a constant increase of the VLSI circuits performances, however the problems related to their testing have multiplied. Nowadays, it is universally acknowledged that test has to be taken into account as soon as possible in the design cycle in order to reduce its cost -and consequently the global electronic component production cost-.

The first Design For Testability (DFT) issues were addressed in the early 1980's, while presently, High Level Synthesis For Testability (HLSFT) approaches are under consideration.

DFT tools are totally independent of synthesis. Working on structural representations at RTL or at gate level, they are applied as post-processors and cannot modify the allocation/binding of physical resources. The 'S-graph' model [1] is commonly used to represent the topology of a sequential circuit to be made testable. At gate level, each node in the S-graph represents a flip-flop (FF) of the circuit. A directed edge from node-1 to node-2 means that the FF corresponding to node-1 is connected to the FF corresponding to node-2 through combinational logic. The sequential depth between two FFs is defined as the number of edges between the two corresponding nodes in the S-graph. When several paths exist between two nodes, the sequential depth is determined by the shortest.

It has been observed ([2], [3]) that the automatic sequential test pattern generation (SATPG) complexity grows exponentially with the length of cycles in the S-graph and linearly with the sequential depth between FFs directly connected to primary inputs and FFs directly connected to primary outputs. Consequently, numerous DFT techniques use these measures to improve testability. For instance, scan FFs are inserted in order to break cycles longer than one and to minimize the sequential depth between inputs and outputs.

Since the early 90's, HLSFT have been proposed for considering testability earlier in the design flow. Because different hardware solutions can meet area/performance constraints, HLSFT systems use criteria associated with specified testability requirements to drive synthesis towards a design which also satisfies testability constraints (see [4] for an overview of University HLSFT systems). Such an approach may avoid the drawbacks inherent in gate-level DFT techniques where test overhead may conflict with initial area and performance requirements.

HLSFT systems target BISTed structures ([5], [6], [7]), or non-scan ([8], [9], [10], [11], [12], [13]) or partial scan designs ([14], [15], [16], [17], [18]). For non-scan and partial scan designs, HLSFT systems attempt to generate structures which can be tested 'easily' by SATPGs. The easiness of testing is evaluated in terms of fault coverage, test efficiency, test sequence length or test generation time. RTL structures are generally extended to gate level descriptions in order to use state-of-the-art SATPGs. Note an exception for the Genesis system [8] where both RTL and gate level descriptions are used to perform hierarchical testing.

Using a S-graph-like model to specify the structure of datapath logic, numerous HLSFT process target the same goals as DFT techniques: break the cycles and minimize the sequential depth at low cost. These systems are aimed at (i) avoiding the formation of cycles and/or restricting the number of scan registers needed to break them ([9], [10], [11], [14], [15], [16], [17], [18]), then at (ii) minimizing the sequential depth between registers directly connected to primary inputs/outputs (I/O registers) ([10], [12], [15], [16], [17], [18]). In [11] and [12], it has also been proposed to reduce the testability overheads by producing designs including as few self-loops (cycles of length one) as possible. But this new constraint may lead the HLSFT system to produce a larger number of cycles longer than one thus lessening testability ([11]). Another goal widely addressed in HLSFT is to generate designs with a maximum number of I/O registers ([9], [10], [11], [15], [18]). Since such registers are easily controlled and observed, they improve the accessibility of the datapath components, and indirectly they entail sequential depth reduction.

Testability constraints can be inserted at several levels of the HLS process. Formation of cycles can be avoided by analyzing the alternatives offered during scheduling or allocation/binding tasks

([9], [11]). Nevertheless, datapath cycles may be either inherent in the behavioral description, or created during the synthesis process (see [16] for an overview). If scan registers are used for breaking remaining cycles in a datapath, their number can be minimized by analyzing the structural implantation alternatives during scheduling and allocation/binding tasks ([17], [18]) or by modifying the behavioral specification before synthesis process ([14]). Scheduling and allocation/binding alternatives are also probed in order to minimize the sequential depth between I/O registers ([12]), and to maximize the number of I/O registers ([10], [11], [12]).

A few comments can be made concerning datapath testability evaluation by means of cycles and sequential depth. First, problems related to cycles and sequential depth are generally less crucial in datapaths than in state machines found in the control logic, due to a 'weaker' interconnect network. Moreover, today commercial SATPGs deal more easily with cycles and sequential depth. Sequential depth minimization can be considered as a second rate objective and cycles have to be distinguished according to the difficulties they involve. For instance in Figure 1.a, there is a path through the adder allowing to control the register R to any particular value from primary inputs. Consequently, in spite of the presence of the cycle, an SATPG can easily find a justification path for a fault in R, in the subtracter, or in some downstream entities. On the contrary, no path allows to control registers R1 and R2 from primary inputs in the cycle of Figure 1.b. This cycle, referred to as 'strongly closed loops' in the rest of this article, can lead to the impossibility of setting necessary test data into R1 or R2 and has to be considered as a testability bottleneck.

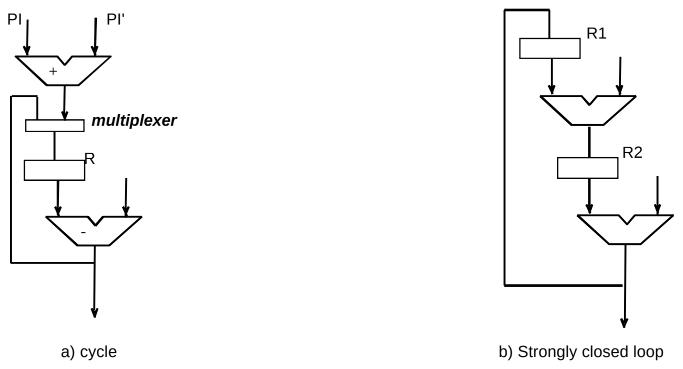

Figure 1. Loops in datapaths

It's worth noting that the difficulty of propagating test data through datapath components is addressed only in few of the works referred above.

Test data propagation problems are addressed only partially and implicitly in HLSFT systems attempting to maximize the number of I/O registers. Essentially, in an ideal binding solution, all registers would be directly connected to primary inputs and outputs. Consequently, the registers and other datapath components connected to them would be fully controllable and observable. Unfortunately, the upper bound of the amount of input and output registers is fixed by the number of DFG variables directly set from primary inputs and directly read through primary outputs. As a consequence, when a maximum number of variables have been assigned to I/O registers, the remainder are assigned into new registers blindly w.r.t. their controllability/observability.

Conversely, the two approaches presented in [8] and [13] are based on testability analysis. In [8], the possibility of transmitting test data to the DFG nodes through arithmetic units and gates are analyzed by a set of rules. For example, to set a test pattern on a multiplier output, an input will be controlled to the value 'one' and the other to the desired test pattern value. Testability analysis results are used to drive the f.u.s/registers binding process. The goal is to generate a datapath structure in such a way that every f.u. in the final structure can be controlled from primary inputs and observed through primary outputs. Control/observation paths, extracted from the DFG paths, form the test environment for the corresponding f.u.s. Test environments are used to propagate test data computed at gate level on f.u.s boundaries. Nevertheless, one limitation of this method is that the set of rules for the testability analysis applies only to specific modules namely, the ones with a neutral element.

In the approach proposed in [13], behavioral modifications for improving testability are applied to hard-to-test sections before HLS. Hard-to-test sections are identified by analyzing the CFG, then variables are classified according to the results of the analysis. For instance, a variable is 'completely controllable' if there exists a sequence of executable paths in the CFG such that, after the execution of these paths, the content of the variable can take any possible value by adjusting the primary input values. A common drawback of both this approach and the one presented in [8] is that the testability analysis is limited to the set of paths already existing in the initial specification namely, in the DFG [8] and in the CFG in [13]. These paths, exercised during the normal execution flow of the circuit, represent only a subset of all the paths connecting datapath components, even though the activation of other paths does not respect the initial sequencing. This point is examined in detail in section III.

Now let us describe the HLSFT system developed at LIRMM for generating datapaths in which as many modules as possible (storage components, f.u.s,... but also muxes and wires) can be tested using parallel test data propagated through primary input/output ports. This system is aimed at exploring the design space to generate easily testable designs while respecting other design constraints, without using partial scan insertion. Test patterns are actually obtained using a gate-level SATPG.

For improving datapath testability, binding/allocation possibilities are explored to enhance the 'accessibility' of all modules. The main features of the presented method are the following:

- It is based on a testability analysis with no restrictive assumptions on the type of f.u.s usable in the designs.
- Each RTL structural path, which exists in the final architecture and that can be derived during the HLS flow, is considered for test data propagation.
- It supports behavioral specifications containing control constructs (with conditional statements and/or cyclic CDFG).
- The strongly closed loops are within the scope of this method (other cycles are not considered as testability bottlenecks).
- Since it deals with datapaths, the sequential depth is considered as a second rate factor allowing to choose between two binding solutions when they are equivalent in terms of other test measures.

- There is no assumption on the value of the test patterns during the HLS process.

II Overview of the system

The primary goal is to take advantage of synthesis possibilities -mainly during the binding phase- in order to obtain a design in which as many modules as possible have their test paths. These test paths being 1/ the controllability paths allowing to apply test patterns on the module's inputs from primary input ports and 2/ the propagation paths allowing to observe test responses through circuit's outputs. Due to the excessive CPU time required by SATPG programs to find test patterns, the secondary goal is to generate designs that are easily testable by SATPG.

The circuit architecture targeted by the synthesis system -depicted in Figure 2- is composed of a datapath and a controller (a set of FSMs) driving the datapath command signals and sequencing the given behavior. Only the testability of the datapath is dealt with in this paper. The controller is assumed to be made testable using a BIST ([19]) or a scan approach. Regarding datapath testability, we consider that any desired value can be set on the control signals, see Figure 2. Therefore, at least two solutions are possible: either the control signals/flags are made fully controllable/observable by using scan method -only to introduce FFs on control signals- or the desired datapath test plan is incorporated within the control logic. Thus the sequencing of the datapath test is independent of the normal mode of execution (contrary to the approaches [8] and [13] discussed in the introduction).

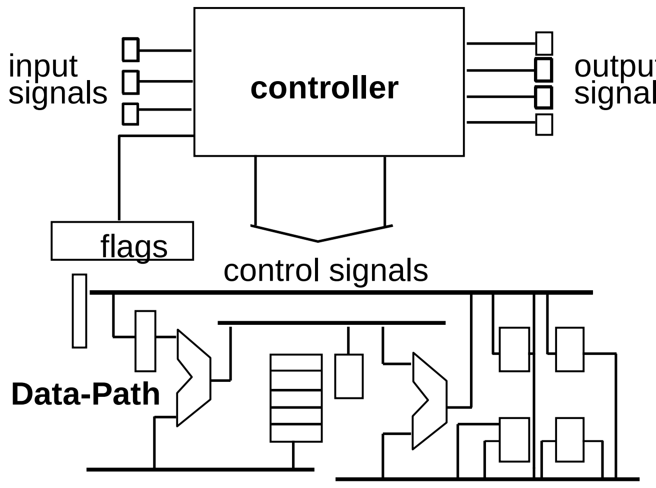

Figure 2. Target architecture

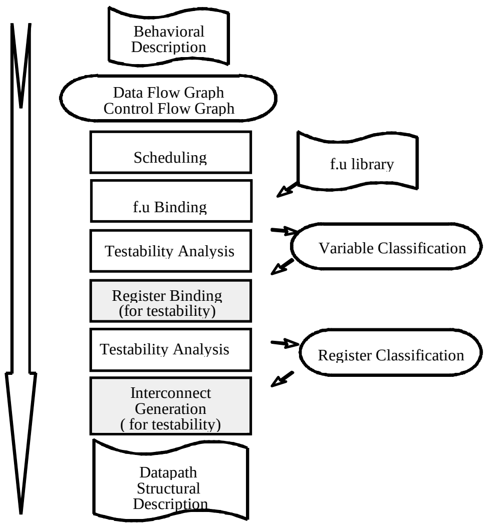
Behavioral Description
Data Flow Graph Control Flow Graph
Scheduling
f.u library
f.u Binding
Testability Analysis
Variable Classification
Register Binding (for testability)
Testability Analysis
Register Classification
Interconnect Generation (for testability)
Datapath Structural Description

Figure 3. Synthesis of testable datapath

The synoptic of the system is shown in Figure 3. White and shadowed boxes are the usual tasks performed during standard HLS of datapath: after the behavioral specification has been translated into a DFG, the scheduling phase assigns operations to control steps. Then operations are bound to f.u.s. Register binding determines the required memory elements (allocation) and binds variables to the registers. Finally, the architectural structure is completed by allocating interconnect elements (muxes, wires, busses) between f.u.s and registers for implementing the data transfers of the initial specification.

To perform HLSFT, several tasks are added and some others modified (stripped and shadowed boxes in Figure 3, respectively). Shaded boxes represent synthesis tasks aimed at improving testability. As shown on the figure, this synthesis for testability method is based on a testability analysis used at different stages of the HLS process. Its results are required for driving register binding and generating interconnections.

Considering the influence of testability improvement on design performance, it is worth noting that:

- the number of control steps stays unchanged since testability is considered after the scheduling phase,
- the clock period may be shortened or lengthened: since from the timing point of view, testability improvement tasks change exclusively the multiplexing resources, delays only through the interconnect network are affected. These multiplexing modifications may change the sum of gates on the critical path. Consequently, the clock period (fixed by the critical path) may be increased or decreased. depending on the sum of gate delays, which is only a rough estimation of the actual delays.

III Behavioral testability analysis

This section deals exclusively with the testability analysis method used before register binding, other invocations of the testability analysis being very alike.

At this point of the synthesis process, circuit's behavior is expressed as a DFG in which operations are bound to f.u.s. Thus, the nodes of the DFG are f.u.s and edges denote data transfers with or without memorization (a variable being associated to each transfer with memorization). Testability criteria analyzed by our system are essentially the existence of 1/ paths allowing to justify test patterns and 2/ paths allowing to observe test responses. Since testability applies to the structural domain, the question is to know whether such paths will exist in the future structure, given the set of data transfers described at the algorithmic level. These paths are to be built from the data transfers regardless of both their grouping into statements and their schedule in the normal flow of execution (contrary to the analysis method presented in [8] and [13]). For instance, if the specification contains a statement like a := b + c, the three transfers a -> +, b -> + and + -> c are considered independently. The future structure will contain a physical path from the registers implementing a and b to the f.u implementing the addition, and from this f.u to the register implementing c. Data transfers are then reorganized in order to create, when possible, justification paths from primary input(s) and propagation paths to primary output(s).

Without loss of generality, we shall assume in the remainder of the text that all the f.u.s have two inputs ports I1 and I2 and one output port O. F.u(x,y)=z means that z is the output value obtained on O if values x and y are the inputs applied on I1 and I2. The term module is used indifferently for f.u.s, variables, registers, muxes. The intrinsic transparency properties of the f.u.s allow to establish test paths. Controllability and observability paths (named C-paths and O-paths) are established for every variable and f.u input/output using the definitions below.

Definition 1: A f.u is *C-transparent* iff $\forall$ z, the value of O, $\exists$ (x,y) respective values of (I1,I2) such that f.u(x,y)=z. Coders are counter-examples of C-transparent modules. In other words a module is C-transparent if any value can be obtained on its output and thus can serve to propagate **any** test pattern to other modules. Other modules like muxes and registers (variables) are C-transparent.

Definition 2: A f.u is *O-transparent* for its input port I1 iff $\forall$ (x,x'), $x \neq x'$, two values on I1, $\exists$ $y_{x,x'}$, a value on I2, such that f.u($x,y_{x,x'}$)=z, f.u($x',y_{x,x'}$)=z' with $z \neq z'$. Usual arithmetic operators are O-transparent if their whole bitwidth is used, while shifters for instance are not. A module is O-transparent if it can differentiate **any** vector pair, in particular a correct test response and a faulty one. Such a module can be used to propagate test responses to the outputs. Muxes, registers and variables are O-transparent.

Definition 3: A *C-path* of a module's port p is a lattice of data transfers rooted on primary inputs and ending at p such that 1/ in every transfer a->b belonging to this C-path, a and b are either variables or ports of C-transparent modules 2/ it does not contain reconvergences, for instance a variable does not feed both inputs of a f.u. A variable or a register with a C-path is said to be *controllable*, it can be set to **any** value. In the same way a f.u is controllable if there exist two distinct controllable variables linked to its inputs (second above condition).

Definition 4: A *O-path* of a module's port p is an ordered set of data transfers starting at p and ending on a primary output, such that for every transfer a->b belonging to the O-path, a and b are either variables, or an input port I1 of a f.u O-transparent for I1, or an output port of O-transparent f.u or a primary output.

Definition 5: To validate a propagation through a O-transparent f.u from I1 to O, a particular value must be justified on I2. Consequently, the propagation is possible if there exist a transfer v->I2 such that v posses a C-path. Such a C-path is called a *lateral C-path*. Thus, any fault on p can be observed (p is said to be observable) if: 1/ x has an O-path, 2/ there exist a complete set of lateral C-paths not containing x. The second condition is set in order to avoid fault masking problems. It is not a necessary condition but the knowledge of gate-level information would be required for its removal. In the remainder of the text, O-path stands for the union of an O-path and its lateral C-paths.

Definition 6: A module is *testable* iff it is controllable and observable.

The extension of the above definitions to actual test patterns is straightforward if a hierarchical testing methodology is envisaged. C-paths and O-paths are a generalization of I-paths and S-paths defined in [20] and [21].

It is important to note that no assumptions are made on test patterns, mainly because this level of design is far from the structural level (registers, muxes, wires have not yet been determined). Thus, this definition of 'testable' is pessimistic. For example, a module may happen to be found as non-controllable by the proposed testability analysis while the actual test patterns can be justified. On the other hand, due to these very conservative assumptions, a SATPG can very easily find test patterns for a module in case it is found testable. Conversely, if the module is determined to be non-testable SATPG may have difficulties either to find test patterns or to prove this module to be actually non-testable.

The testability analysis process returns classification of variables depending on their respective controllability/observability. Moreover, when a variable is found either controllable or observable, corresponding C-paths or O-paths are generated, respectively. The sequential depth of a controllable variable is defined as the minimum number of clock cycles required to set a test pattern in this variable. For example, the sequential depth of the controllable variables b, c, d, e and i, shown on the DFG of Figure 4, is equal to 1, while the controllable variable k has a sequential depth equal to 2.

This paragraph describes the classification of non-controllable variables (non-observable variables can be classified using a similar approach). The example of Figure 4 will serve as a support for illustrating this classification. Non-controllable variables are iteratively clustered into three classes. These classes are successively built up starting from (1) the class of non-controllable variables because they are only fed by not C-transparent f.u.s (e.g. variables h, a), continuing by (2) the class of variables that would become controllable if some of the variables of the previous class were made controllable (e.g. variables g, j), and ending by (3) the class of remaining variables. This last class contains first, (3.1) the variables belonging to strongly closed loops (e.g. variable f) and second, (3.2) the variables that would become controllable if some of the variables of (3.1) were made controllable (e.g. variable l). This allows the set of strongly closed loops to be limited to the those containing only transparent f.u.s. The variables belonging to strongly closed loops containing non-transparent f.u.s are included in the first two classes (e.g. variables a, g) since even if such loops were open the concerned variables would still remain non-controllable.

Let us recall that the cycles considered as testability bottlenecks are only the strongly closed ones. For example, the cycle involving the variable a in the DFG shown in Figure 5 is not of the 'strongly closed loop' one. This variable can be set from primary inputs through functional unit f.u.1 which implements both +1 and +2 operations and is considered as a controllable variable.

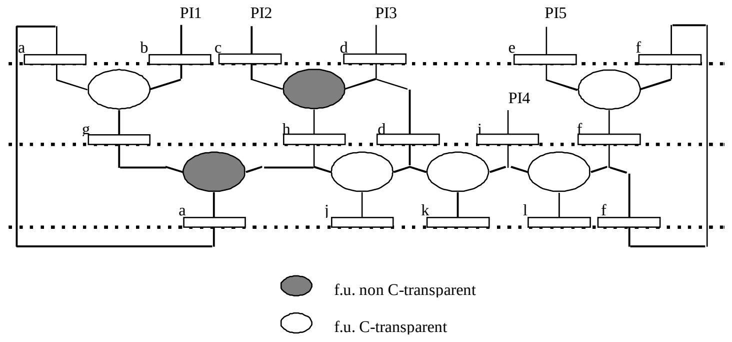

Figure 4. DFG with strongly closed loop

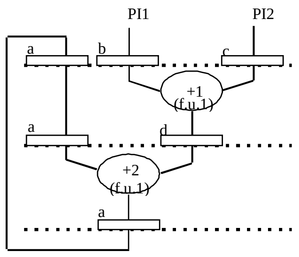

Figure 5. DFG without strongly closed loop

C-paths and O-paths are actually determined iteratively by scanning the list of elementary data transfers and checking the transparency of the involved f.u.s. Concerning C-paths, initially the variables linked to primary inputs are controllable, their C-path containing only one transfer. Let us assume that after several iterations, the data transfer from a C-transparent f.u. F to a variable V is under consideration. If both F inputs have been found controllable during previous iterations, and if their respective C-paths end on different variables (see Def3 about reconvergence), then V is controllable and its C-path is the union of the two C-paths plus the transfers F->V. This C-path is added to the list of C-path for V. The process is iterated until no change occurs. The interested reader can refer to [22] for more details on the testability analysis algorithm.

Figure 6 shows the scheduled DFG obtained for the 'differential equation' high level synthesis benchmark [23]. Operations have been bound to two multipliers MULT1 and MULT2, one adder ADD and one subtracter SUB. The DFG is cyclic, feedback edges are not represented on the figure. Variables x, y and u are loaded from primary inputs in the first iteration, then loaded from the f.u.s in the following iterations. For illustration purposes, we assumed that all f.u.s are C-transparent and O-transparent except the subtracter which actually is transparent.

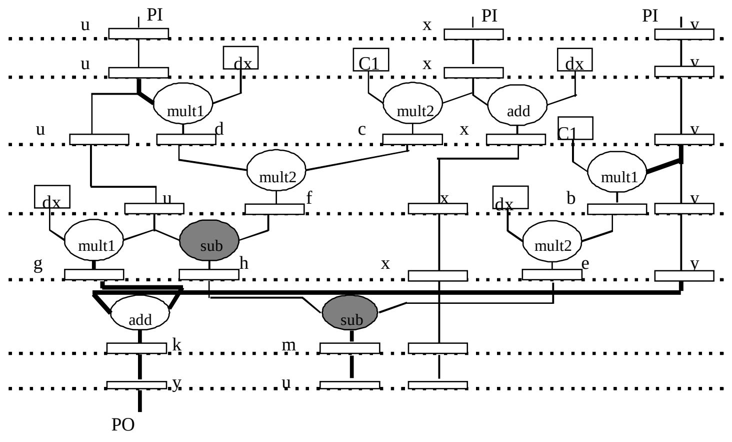

Figure 6. Scheduled DFG

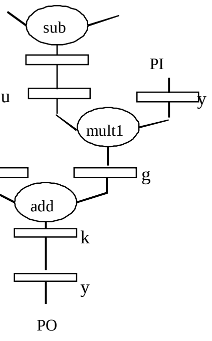

Figure 7. Propagation path for a fault in SUB

Figure 7 represents a propagation path for a fault occurring in SUB. It is built up using the transfer from SUB's output port to the observable variable m, and the O-path of this variable. This O-path has been established using the elementary transfers illustrated by thick lines in Figure 6. The lateral C-paths are PI->y->mult1 and PI->y->add.

Table 1 shows the results of the testability analysis on this example. Rows 2 and 3 respectively assess the controllability/observability of the variables. Rows 2 and 3 in Table 2 give the number of possible justification paths for each input of the f.u.s. Row 4 (Table2) gives the number of propagation paths for each f.u.

Table 1 : Testability characteristics of variables

var.	y	x	u	b	d	k	e	f	c	h	m	g
cont.	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	no	yes
obs.	yes	yes	yes	no	no	yes	no	no	no	no	yes	yes

Table 2 : Testability characteristics of functional units

	Mult1	Mult2	Add	Sub
input 1 (just. paths)	1	1	2	1
input 2 (just. paths)	1	1	1	4
output (prop. paths)	1	0	1	1

IV Synthesis algorithms

According to analysis results, testability is considered during the synthesis steps represented by shadowed boxes in Figure 3 namely, the register binding phase and the generation of the interconnect network (insertion of busses, muxes and wires necessary to interconnect registers and f.u.s). Allocation and binding choices are sought in order to improve testability. Only register allocation for testability improvement is detailed here, the principle being the same as for generating the interconnect network.

a- Register allocation:

Let us first recall the basic principle of register allocation: variables can share the same register if their life times do not overlap. The merging possibilities of variables are extracted by performing a life time analysis and a compatibility graph is built up. In this graph, nodes and edges stand for variables and merging possibilities, respectively. Every clique of the graph represents a set of variables which can be assigned to the same register.

In standard HLS, a solution for minimizing the area expressed by a cost function f_a (# registers, # multiplexer inputs,...) is reached by seeking an adequate clique partitioning in the compatibility graph. Generally, the search is guided by weighting the graph edges by the area gain Df_a -i.e. the area gained by merging the two involved variables-. In the algorithm used in our standard HLS system [24], the pair of variables (registers) joined by the edge of maximal weight is merged. After each coupling, all weights and compatibility edges are updated. The process is iterated until no further coupling is possible.

The principle underlying testability improvement is to assign the same register to non-controllable (non-observable) variables and to controllable (observable) ones rather than assign only non-controllable variables to the register. In the first case, the resulting register is controllable while in the second case it is not. Registers inherit the testability properties of variables assigned to them, e.g. the C-path (O-path) of a controllable (observable) variable becomes the C-path (O-path) of the register. Figure 8 illustrates the principle underlying register allocation for testability improvement.

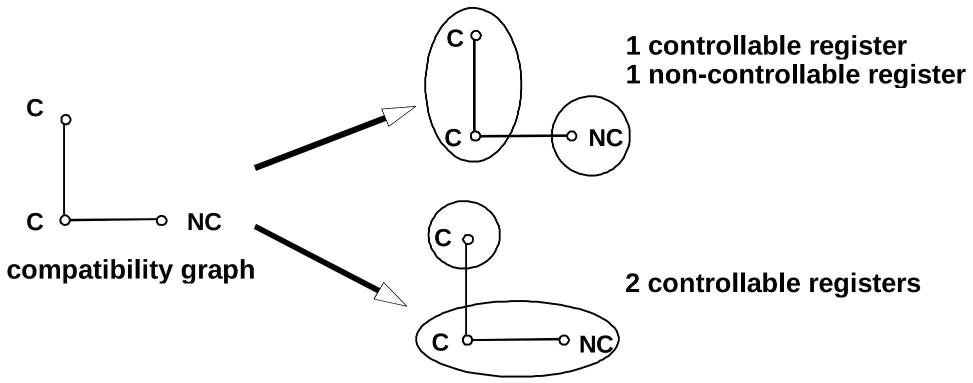

Figure 8. Register allocation for testability : principle

Figure 9 shows how register allocation affects not only the testability of the registers but also of f.u.s, muxes and wires (only controllability is illustrated). A partially hardware mapped algorithmic specification is given in a). Assuming that F.U.1 is not C-transparent, all variables except a and b are non-controllable. Two different register allocations are given in b) and c) together with the resulting architectures. A point-to-point style of interconnection has been used to interconnect the modules. The number of non-controllable modules (black boxes) and connections (bold lines) is drastically reduced in solution c) compared to b). The right input of F.U.2 is fed only by registers fed in turn by F.U.2 in b) making this loop non-controllable. In c), the C-path "input->R2->R5" allows to load this f.u input with any test pattern. The association of variable b with f, instead of g, breaks the loop of case b). This kind of possibility is exploited during register allocation for testability. In both designs, register R3 is non-controllable because it is only fed by F.U.1.

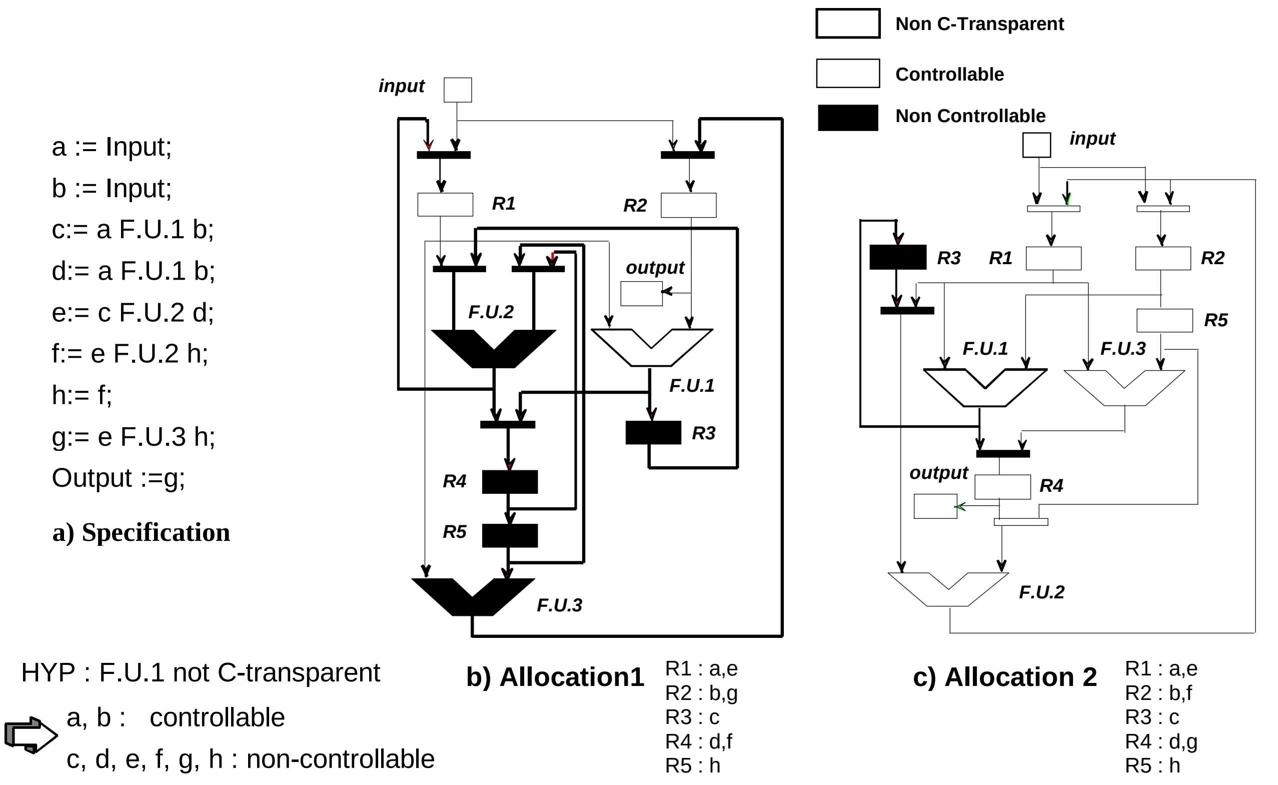

Figure 9. Standard register allocation vs register allocation for controllability improvement

The proposed method does not differ from other related works in the general principle but in the objectives targeted to improve testability. As in many other HLSFT systems, the rule is to assign non-testable behavioral components together with testable ones -since structural components resulting from the assignations inherit testability properties-.

Concerning the objectives, the problem to solve is how to bind variables onto registers in such a way that as many structural components as possible would possess C-path and O-path. Consequently, breaking cycles (except strongly closed loops) is not targeted, and the sequential depth is only a second rate factor allowing to choose among different assignations of like interest w.r.t. C-paths and O-paths establishment.

To take into account testability as well as area, Df_a is replaced by $aDf_a / Df_{amax} + b(Df_c/Df_{cmax} + Df_o/Df_{omax})$ where Df_c (resp. Df_o) is the controllability (resp. observability) gain, Df_{amax}, Df_{cmax} and Df_{omax} are normalization factors. a and b are user defined tuning factors allowing tradeoffs between area and testability. The controllability gain is intended to measure the effect of a merge on the controllability of the whole circuit. As mentioned earlier, merging a non-controllable variable with a controllable one into a register makes this register and the downstream f.u input port connected to it by a transfer both controllable. Therefore, controllability gain is expressed as a linear combination of:

- the number of f.u.s that this merge would make controllable,
- the number of f.u.s' inputs made controllable while the f.u remains non-controllable,
- the number of registers made controllable,
- the difference of sequential depth between these two registers,
- a priority factor taking into account the controllability nature of the two registers. For instance , a high coefficient is given if this coupling breaks a sequential loop.

Observability gain is computed in a similar way. The interested reader can refer to [25] for details on gain computation.

Table 3 gives the results of register allocation for the Differential Equation example after an allocation taking no account of testability (i.e. b=0) and an allocation including testability (a=b=1) using the above algorithm.

Table 3 : Comparison of registers controllability/observability

Without testability				With testability			
var.	Reg.	cont.	obs.	var.	Reg.	cont.	obs.
u	Reg1	yes	yes	u	Reg1	yes	yes
x	Reg2	yes	yes	**x**	Reg2	yes	yes
y	Reg3	yes	yes	**y**	Reg3	yes	yes
b,k	Reg4	yes	yes	b,m	Reg4	yes	yes
d,g	Reg5	yes	yes	d,h	Reg5	yes	yes
e,c	Reg6	yes	yes	k,f	Reg6	yes	yes
f,m	Reg7	yes	yes	e,c	Reg7	yes	yes
h	Reg8	no	no	g	Reg8	yes	yes

In the first case, register Reg8, assigned only to variable h, is neither controllable nor observable. In the second case, each register is controllable and observable. Consequently any test pattern can be justified on primary inputs and any test response can be observed on the primary outputs, from each f.u and register ports. The pseudo-code describing the possible transfer sequence for testing the MULT1 f.u (its C-Path and O-Path) is given below:

```
Reg1:=PI(TP1);        /*TP1, TP2: test patterns pre-computed by a combinational ATPG to test a fault in MULT1*/
Reg3:=PI(TP2);
Reg5:=MULT1(Reg1,Reg3);                                   /*Reg5 memorizes MULT1's test response (TR)*/
Reg2:=PI(1);                        /*control of the lateral input of the O-transparent f.u MULT2 with the value 1*/
Reg6:=MULT2(Reg5,Reg2);                                               /*TR propagation through MULT2*/
Reg3:=Reg6;
PO(TR):=Reg3;                                                       /*TR observation on primary output*/
```

It must be underlined that register allocation can only enhance -but not guarantee- the testability of the circuit since it is constrained by the register sharing possibilities.

b- Interconnect network generation:

If a wiring model allowing to share interconnections is chosen (e.g. a bus based model), hardware sharing possibilities can also be sought either to enhance or to guarantee testability. In the first case, an interconnect network maximizing the number of testable points is generated while keeping the number of connections to a minimum. In the second case, all the points are made testable while adding as few extra connections as possible. Arguments and method are similar to those used for register allocation.

V Results

This test/synthesis method has been applied to four behavioral synthesis benchmarks. Their characteristics before register allocation are given in Table 4. Tseng's example is borrowed from [26], the differential equation example from [23], the AR filter from [27] and the elliptical filter (EW) from [28]. Columns 2 to 7 show respectively the number of control steps, f.u.s, variables, constants and primary inputs outputs. Columns 8 and 9 give the number of controllable and observable variables.

Table 4: Benchmarks characteristics

Example	# steps	# f.u.	# var	# constants	# Primary Inputs	# Primary Outputs	# c. var.	# o. var.
tseng	13	2+,2-	11	0	2	1	5	3
differential	9	1+,1-,2*	12	2	3	1	10	6
ar filter	21	1+,2*	20	2	4	2	4	6
ew filter	19	2+,1*	39	3	1	1	2	1

All the designs have a constant bitwidth. Unfortunately, in the first two HLS examples, the operations are only additions, subtractions and multiplications which do not demonstrate our purposes as they are implemented by transparent f.u.s. For illustration purposes on transparency, we replaced the subtracters by modules which are neither C-transparent nor O-transparent -actually we replaced the "subtracters" by "multipliers" where only the n/2 most significant bits are used-. In the two filters, multipliers are used for implementing multiplications by constants. As a consequence, and according to the assumption on controllability, there is no way to make their outputs and downstream registers controllable. Thus, some of the f.u.s of these examples present transparency bottlenecks at behavioral level and at gate-level as well.

The examples were synthesized with point-to-point style interconnections. A one level of multiplexers network was used to connect the modules. In this model, no connection sharing possibilities can be used to enhance testability. The improvement is only due to register allocation. Area comparisons are reported in Table 5 while testability results are given in Table 6 where the two lines associated to each benchmark correspond respectively to a synthesis style without testability constraint (w/o test) and with testability constraint (with test). Columns 3 to 7 in Table 5 show respectively the number of registers, multiplexers, multiplexers' inputs, wires and point-to-point wires. It must be noticed that areas are roughly the same.

Table 5: Area comparison

Design	synthesis style	# reg	# mux	# mux in.	# wires	# wire fanout
tseng	w/o test	5	7	17	18	67
	with test	5	10	24	21	81
differential	w/o test	5	9	23	23	79
	with test	5	11	30	25	93
ar filter	w/o test	9	9	27	27	97
	with test	8	12	36	29	112
ewf filter	w/o test	12	8	36	25	122
	with test	11	11	54	27	155

In Table 6, columns 3, 5, 7 and 9 give respectively the number of non controllable (NC) registers, f.u.s, wires and mux. inputs. Columns 4, 6, 8, 10 give the number of non observable (NO) registers, f.u.s, wires and mux. outputs. The last two columns give respectively the ratio of testable registers and f.u.s.

Table 6: Designs testability (RT level)

Design	synthesis style	NC reg.	NO reg.	NC f.u	NO f.u	NC wires	NO wire fanouts	NC mux inputs	NO mux outputs	Testable registers	Testable f.u.s
tseng	w/o test	2	1	2	0	14	25	18	6	3 / 5	2 / 4
	with test	1	1	0	0	6	27	12	8	4 / 5	4 / 4
differential	w/o test	0	1	0	1	6	23	12	4	4 / 5	3 / 4
	with test	0	0	0	0	6	8	14	2	5 / 5	4 / 4
ar filter	w/o test	5	5	3	1	28	59	38	12	0 / 9	0 / 3
	with test	0	0	2	0	12	24	12	4	8 / 8	1 / 3
ewf filter	w/o test	1	11	1	2	6	115	6	16	0 / 12	0 / 3
	with test	0	0	1	0	4	14	12	2	11 / 11	2 / 3

In order to improve the evaluation of the designs produced by our behavioral synthesis system and to compare actual testability criteria with ours, we expanded our RT-designs into gate-level designs and used the commercial SATPG tool Sunrise[29]. Table 7 gives SCOAP based testability results at gate-level. Column 4 gives the number of nodes with SCOAP testability measure between 0 and 10, column 5 gives the number of nodes with SCOAP testability measure between 10 and 50, etc. ATPG results are reported on the right hand side columns. The results clearly show a good correlation between the metrics we used to orient behavioral synthesis and the actual testability of the designs and the reduction of the number of non testable faults.

Table 7 : Designs testability (gate-level)

		Testability measures						Sequential ATPG results	
Design	Synthesis style	# Nodes	10	50	100	500	∞	fault coverage for 100% efficiency	ATPG CPU time (in sec.)
tseng	w/o test	437	0	349	49	0	39	95.18	11,69
	with test	501	0	446	26	0	29	96.98	11,66
differential	w/o test	558	0	322	107	41	88	85.35	41.68
	with test	638	0	330	248	11	49	94.29	34.23
ar filter	w/o test	792	0	407	15	0	370	56.87	4.68
	with test	849	0	526	62	0	261	66.99	12.68

ewf filter	w/o test	1041	0	245	29	0	767	25.68	27.80
	with test	607	0	340	217	2	48	87.44	22.53

In the previous experiments, the examples present relatively few design alternatives. Furthermore, a 100% efficiency is easily obtained due to their small size. In order to demonstrate the sensitivity of our test/synthesis algorithms to user defined constraints (ratio b /a), we have generated larger designs with a higher sequential depth. Therefore, on the same benchmarks, we used registers made with latches instead of flip-flops which lead to larger designs since the compatibility graph is less dense. We kept the same set of f.u.s as before except for the ewf filter, which had three adders instead of two and three multipliers instead of one. The designs were synthesized for various (b / a) values. Results are reported in Table 8. For each example, "Original" denotes the synthesized circuit without testability considerations (b=0), while "Testsyn" et "Testsyn1" represent versions in which the ratio testability/area (b /a) was respectively set to 100 and 1.

Table 8 : Area and testability estimations of designs at RT level

		Area					Testability							
Design		reg	mux	mux inputs	wires	wire fanout	NC reg.	NO reg.	NC f.u.s	NO f.u.s	NC wires	NO wire fanouts	NC mux inputs	NO mux outputs
tseng	Original	7	10	22	23	38	2	4	2	1	5	26	7	7
	Testsyn	7	11	24	24	40	2	0	1	0	4	9	7	2
differential	Original	8	13	29	30	46	1	1	0	0	4	7	7	2
	Testsyn	8	14	32	31	49	0	0	0	0	3	6	7	2
ar filter	Original	10	9	28	28	46	6	6	3	1	10	31	14	7
	Testsyn1	10	11	32	30	50	0	2	2	0	4	11	4	2
	Testsyn	9	11	31	29	48	0	1	2	0	4	9	4	2
ewf filter	Original	19	13	52	42	84	4	18	3	5	9	81	8	13
	Testsyn1	18	14	60	42	91	3	0	3	0	8	10	7	2
	Testsyn	17	21	81	48	111	1	0	3	0	6	9	5	2

It's well worth noting that the testsyn designs' testability was drastically improved while the area was kept roughly the same moreover, testability evolved in the same direction as the ratio b / a as was expected.

We expanded four of these designs to gate-level and ran sequential ATPG on them. The results are reported in Tables 9 and 10. The second column indicates the maximum allowed CPU time per fault. The number of aborted faults is computed on the reduced fault list (i.e. after fault equivalence checking).

Table 9: ATPG results: EW filter (Total faults: Original = 3768, TestSyn = 3552)

Design	CPU limit per fault	Detected faults	Potentially detected	Aborted faults	"Untestable" faults	Testable Faults Coverage	Test Efficiency	CPU (s)
Original	1 sec	3213	28	63	489	97.98	98.24	462
TestSyn	1 sec	3166	35	37	349	98.84	98.95	116
Original	100 sec	3228	6	25	511	99.1	99.23	10573
TestSyn	100 sec	3171	0	0	381	100	100	726
Original	200 sec	3197	3	14	514	99.5	99.57	19691
TestSyn	200 sec	3171	0	0	381	100	100	726

Table 10: ATPG results: Differential equation (Total faults: Original = 1926 , TestSyn = 2022)

Design	CPU per fault	Detected faults	Potentially detected	Aborted faults	"Untestable" faults	%Testable Faults Coverage	Efficiency	CPU (s)
Original	1 sec	1252	0	328	177	71.58	74.19	478.8
TestSyn	1 sec	1650	0	114	181	89.62	90.55	402.3
Original	100 sec	1739	0	6	181	99.65	99.68	3372
TestSyn	100 sec	1819	0	6	185	99.02	99.1	1614
Original	300 sec	1740	0	4	182	99.77	99.79	5003
TestSyn	300 sec	1835	0	1	186	99.94	99.95	2237

This experiment demonstrates that the designs obtained by taking testability into account are far more easy to test than those without testability. This table shows that the CPU time, to obtain the same ATPG efficiency, is considerably shortened for the testsyn designs.

Finally we compared this HLSFT method with a classical DFT method (namely partial scan). On a collection of examples, we used the automatic partial scan chain extraction tool from the Sunrise suite [29]. This tool is ATPG driven. We used this tool on "original" designs and constrained it to achieve the same fault coverage as in the testsyn designs.

Firstly, the area overhead (as estimated with the Synopsys [30] on 1.0. micron technology)) is on average 5% for testsyn designs versus 9% for partial scan designs (for information, the full scan versions of original designs present an average area overhead of 19.5%).

Secondly, considering test application time, the average ratio (time for partial scan design / time for testsyn design) is 24.93 with a maximum of 58.41. Nevertheless, on one example, test application time is a little bit shorter for the partial scan design than for the testsyn one (ratio 0.9).

Finally, concerning CPU time, the scan chain extraction requires several additional CPU hours (or even days) due to the necessity of SATPG runs, while HLSFT needs just a few seconds more than standard HLS.

VI Conclusion and directions for future research

The methods presented here demonstrate how the design testability can be questioned and improved during behavioral synthesis. Such an approach may avoid or at least simplify the prevalent approach where testability issues are considered at gate level as it is based on a testability analysis method working at different abstraction levels of the circuit specification. Two synthesis tasks (register binding and interconnect generation) have been explored for taking into account testability criteria. Results show a drastic improvement both on the testability of the final circuit and on the effectiveness of SATPG.

Experiments show that this HLSFT method is a valuable alternative to classical gate-level DFT techniques. From the present version of the system, a number of improvements and extensions are possible.

Presently, the coefficients used during register allocation -area and test weights as well as coefficients used in the linear function of controllability and observability gain- are static. In a future version, these coefficients will be dynamically tuned depending on the remaining testability problems, for instance it may be misleading to give a non null coefficient to controllability gain while the remaining problems concern observability.

Functional units allocation and binding as well as scheduling will also be questioned since they affect the number and nature of data transfers from which testability is predicted.

As long as a point is identified to be testable by the testability analysis method (in which case justification and propagation paths are given), there is no need to use a SATPG to find test patterns and to generate a test plan. Future research will be devoted to the study of the quality of such a generated test plan and its implementation within the controller.

Finally, the extension of this method to partial scan is currently under development which will imply new modifications of the allocation/binding algorithms.

Acknowledgment

The authors wish to thank the reviewers for their suggestions and comments.

References

[1] K.T. Cheng, V.D. Agrawal: “Synthesis of testable finite state machine”. Proc. ISCAS, pp: 3114-3115, May 1990.

[2] K.T. Cheng, V.D. Agrawal: “A Partial Scan Method for Sequential Circuits with Feedback”, IEEE Transactions on Computers, 39, pp 544-548, April 1990.

[3] D.H. Lee, S.M. Reddy: “On Determining Scan Flip-Flops in Partial Scan Designs”, proc. ICCAD, pp 322-325, 1990.

[4] L. Avra, E.J.McCluskey: “High-Level Synthesis of Testable Design: An Overview of University Systems”, proc. ITC, Test Synthesis Seminar, Digest of Papers, pp 1-8, 1994.

[5] I.G. Harris, A Orailoglu: “Micro-architectural Synthesis of VLSI Designs with High Test Concurrency”, proc. DAC, pp 206-211, 1994.

[6] C. Papachristou, S. Chiu, H. Harmanani: “SYNTEST: a method for high-level SYNthesis with self-TESTability”, proc. ICCD, pp 458-462, 1991.

[7] L. Avra: “Allocation and Assignment in High-Level Synthesis for Self-Testable Data-Paths”, proc. ITC, pp 463-472, 1991.

[8] S. Bhatia, N.K. Jha: “Genesis: A Behavioral Synthesis for Hierarchical Testability“, proc. ED&TC, pp: 272-276, 1994.

[9] A. Mujumdar, R. Jain, K. Saluja: “Behavioral Synthesis of Testable Designs”, proc. FTCS, pp: 436-445, 1994.

[10] T.-C. Lee, W. H. Wolf, N. K. Jha, J. M. Acken: “Behavioral Synthesis for Easy Testability in Data Path Allocation” proc. ICCD, pp. 29-32, 1992.

[11] A. Mujumdar, R. Jain, K. Saluja :“Incorporating Testability Considerations in High-Level Synthesis”, IEEE J. of Electronic Testing, pp 43-55, Feb. 1994.

[12] T. Kim, K-S Chung, C.L. Liu: “A Stepwise Refinement DataPath Synthesis Procedure for Easy Testability”, proc. ETC, pp586-590, 1994.

[13] C-H. Chen, T. Karnik, D.G. Saab: “Structural and Behavioral Synthesis for Testability Techniques”, IEEE Trans. CAD, vol. 13, no 6, pp 777-785, June 1994.

[14] S. Dey, M. Potkonjak: “Transforming behavioral specifications to facilitate synthesis of testable designs”, proc. ITC, pp 184-193, 1994.

[15] T-C. Lee, N.K. Jha, W.H. Wolf: “Conditional resource sharing method for behavioral synthesis of highly testable data paths”, proc. ITC, pp 744-753, 1993.

[16] S. Dey, M. Potkonjak, R. Roy: “Exploiting Hardware Sharing in High-Level Synthesis for Partial Scan Optimization”, proc. ICCAD, pp 20-25, 1993.

[17] M. Potkonjak, S. Dey, R. Roy: “Behavioral synthesis of Area-Efficient Testable Designs using interaction between Hardware Sharing and Partial Scan”, IEEE Transactions on CAD, vol. 14, no 9, pp 1141-1154, Sep. 1995.

[18] T.C. Lee, N. Jha, WH Wolf: “Behavioral Synthesis for Highly Testable Data Paths under Non-Scan and Partial-Scan Environments”, proc. DAC, pp 292-297, 1993.

[19] S. Hellebrand and H-J Wunderlich: “Synthesis of Self-Testable Controllers”, proc. ED&TC, pp 580-585, 1994.

[20] M.S. Abadir, M.A. Breuer: “A knowledge-based System for Designing Testable VLSI chips”, IEEE Design&Test, pp 56-68, Aug. 1985.

[21] S. Freeman: “Test Generation for Data-Path Logic: The F-path method”, IEEE Journal of Solid State circuits, vol. 23, no 2, pp 421-427, April 1988.

[22] D. Hammad: “Testability and architectural synthesis of digital circuits”, PhD Dissertation (in french), Université de Montpellier, 1995.

[23] P.G. Paulin, J.P. Knight: “Scheduling and binding algorithm for high level synthesis”, proc. DAC pp. 1-6, 1989.

[24] B. Rouzeyre, G.Sagnes: “A new method for the minimization of memory related area”, proc. EURO-ASIC 91, pp 184-189.

[25] M.L. Flottes, D. Hammad, B. Rouzeyre: “High Level Synthesis for Easy Testability”, proc. ED&TC, pp. 198-206, 1995.

[26] C.-J. Tseng and D. Siewiorek: “Automated Synthesis of Data Paths in Digital Systems”, IEEE Transactions on CAD, vol.5, no 3, July 1986.

[27] R. Jain, K. Kucukcaker, M.J. Mliner, and A.C. Parker: “Experience with ADAM synthesis system”, proc. DAC, pp 56-61, 1989.

[28] P. Dewilde, E. Deprettere, R. Nouta , “Parallel and pipelined VLSI implementation of signal processing algorithms”, In VLSI and Modern Signal Processing. S.Y.Kung, H.J.Whitehouse, T.Kailath Editors. Prentice Hall. pp: 257-260.

[29] Sunrise tests system: “Testgen” reference manual. Version2.1, November 1994.

[30] Synopsys reference manual : “design analyzer”. Version 3.0.a, December 1992.