Optimizing Construction of Scheduled Data Flow Graph for On-line testability

Bernard Kamsu-Foguem, Emmanuel Simeu

OPTIMIZING CONSTRUCTION OF SCHEDULED DATA FLOW GRAPH FOR ON-LINE TESTABILITY

Bernard Kamsu-Foguem[a,*], Emmanuel Simeu[b]

[a]Laboratory of Production Engineering (LGP), EA 1905, ENIT-INPT University of Toulouse, 47 Avenue d'Azereix, BP 1629, 65016, Tarbes Cedex, France
[b]TIMA-Laboratory/ University of Grenoble, 46, Avenue Félix Viallet, F-38031 Grenoble, France

ABSTRACT
The objective of this work is to develop a new methodology for behavioural synthesis using a flow of synthesis, better suited to the scheduling of independent calculations and non-concurrent online testing. The traditional behavioural synthesis process can be defined as the compilation of an algorithmic specification into an architecture composed of a data path and a controller. This stream of synthesis generally involves scheduling, resource allocation, generation of the data path and controller synthesis. Experiments showed that optimization started at the high level synthesis improves the performance of the result, yet the current tools do not offer synthesis optimizations that from the RTL level. This justifies the development of an optimization methodology which takes effect from the behavioural specification and accompanying the synthesis process in its various stages. In this paper we propose the use of algebraic properties (commutativity, associativity and distributivity) to transform readable mathematical formulas of algorithmic specifications into mathematical formulas evaluated efficiently. This will effectively reduce the execution time of scheduling calculations and increase the possibilities of testability.

Keywords
High-Level Synthesis, Expression Evaluation, Algebraic Transformations, Scheduling, Testability.

1. INTRODUCTION
The high-level synthesis is defined as a sequence of successive refinements to transform a specified behaviour in a Hardware Description Language into a list of interconnected numerical operators or logic gates. More formally, behavioural synthesis is an automated design process that interprets an algorithmic description of a desired behaviour and creates hardware that implements that behaviour [2]. It corresponds to the passage from the behavioural domain to the structural domain. High-level synthesis tools seek, first, to compile a behavioural view of a target model suitable for the synthesis and independent of the used language. Then, they optimize the model before making a structural projection on an abstraction level immediately below.
Thus, there are several levels of synthesis and each synthesis step can, therefore, to refine the granularity of the description of the circuit, and at the expense of its complexity. The final description takes the form of drawing plans masks defining the circuit topology [3]. To achieve this goal, several steps corresponding to different modelling levels are presented in logical sequence, including descriptions of circuit behaviour, architecture, logical structure (Boolean model with two values 0 and 1, modelling the voltages across the circuit), electrical structure (assembly and sizing of transistors) and topological structure (drawing plans masks respecting strict geometric constraints) [5].
The online testing provides the ability to detect any occurrence of a failure in a circuit without affecting its normal operation; particularly the non-concurrent online testing contributes to safe operation of integrated circuits for the realization of the operational parts self-controllable [8]. This technique consists in anticipation of faults by modifying the scheduling of data flow graph to insert the test operations of functional units during their rest periods. In short, it is a modification of scheduled data flow graph with idle-time utilization for on-line testability [11]. While concurrent fault detection is mainly achieved by hardware or software redundancy, like duplication, non-concurrent fault detection, particularly useful for periodic testing, is usually achieved through hardware-based self-test [6].
Note that two main criteria for judging the quality of a synthesis tool, namely: the expressiveness of its input language and powerful of its optimization methods. Also, whatever the level of abstraction, synthesis takes into account different optimization objectives such as minimizing the delay, the surface, the circuit consumption, and more recently testability. However, current tools do not provide an appropriate optimization of high-level synthesis, hence the need to develop an optimization methodology which takes effect from the behavioural specification and accompanying synthesis process throughout its various stages [7]. This will produce better results through improved scheduling and testability.
The interest of this work is to optimize the evaluation of algebraic expressions in behavioural synthesis. This assessment is at the heart of most scientific computing applications. For example, modelling of any physical phenomenon results in a construction of a mathematical equation, and this is then interpreted into the form of one or more formal expressions in the computer program. Generally, thanks to the algebraic properties such as associativity and commutativity of operators, there are several ways to write or evaluate an expression. Moreover, the evaluation time of two mathematically equivalent expressions can be significantly different. Speed of the program is strongly linked to the effectiveness evaluation of these expressions. Because of the algebraic properties of operators handled, we propose to introduce a technique for rewriting expressions into equivalent mathematical forms, but better suited (as containing less dependencies) with a parallel instruction execution, including the scheduling and the online test in the high level synthesis.
*

* Corresponding author. Email: Bernard.Kamsu-Foguem@enit.fr

2. BEHAVIOURAL SYNTHESIS AND TESTABILITY

The behavioural synthesis also called high-level synthesis, is the transformation of an algorithmic description into a Register-Transfer Level (RTL) description. More concretely, the high-level synthesis is a sequence of tasks that transform a data flow graph or control flow graph in a more detailed description called register transfer level. This description gives the two major components forming a circuit, namely the processing unit and control unit. The processing unit, also called data path, contains the functional units and registers. The control unit determines at each clock cycle which operations of the data flow graph or control flow graph must be performed and by which functional units. The objective of the high-level synthesis is to find a digital circuit that satisfies a given specification in the form of an algorithm, but from a behavioural description, it is possible to generate a large number of architectural solutions, at the heart of the problem is therefore to obtain the best architecture within the constraints imposed [1]. The first stage of the high-level synthesis usually involves the compilation of the algorithmic description into an internal representation.

At the highest level of abstraction, the specification describes the functionality of the circuit and without regard to hardware or other implementation details. The specification is given in a hardware description language such as VHDL. It is then translated into a graph representation called the Data Flow Graph (DFG). In its simplest form DFG is a directed graph whose nodes represent atomic operations that the system must perform, and the arcs represent the precedence between operations which are due to data dependencies. The DFG will serve as an input model for the scheduling step.

The high-level synthesis is composed of four interdependent tasks [14]: (1) the selection of functional units, (2) the scheduling of the operations of DFG, (3) registers allocation to variables, and (4) the bus allocation to data transfers.

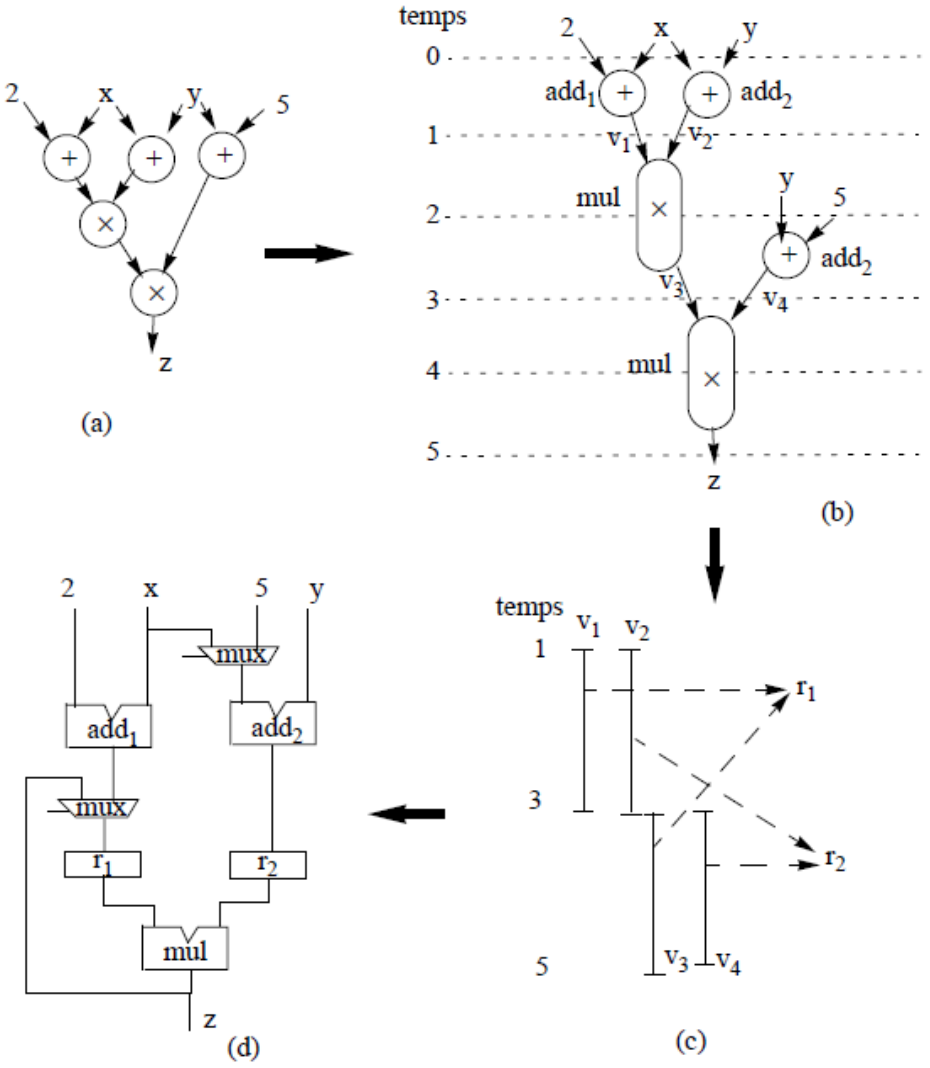

Figure 1. the different steps of a behavioural synthesis:
(a) The DFG expression (b) Scheduling (c) The Time to Live of variables with a registers allocation (d) Data path obtained after connections synthesis

2.1 Scheduling operations of DFG

In what follows, we will focus on the step of scheduling operations of DFG. The objective of this task is to determine a static execution order of operations of DFG by the functional units. The order must respect the dependencies between operations. There are usually two types of scheduling: scheduling under resources constraints and scheduling under performance constraints. In the first category, the maximum number of functional units for use in the circuit is fixed in advance, and the objective is to minimize the

duration of the scheduling. In second category, the duration of the scheduling is fixed, and the objective is to minimize the number of functional units used.

Example: suppose we have two adders {add1, add2} and a multiplier (MUL). The duration of an addition is a unit of time and the duration of a multiplication is two units of time. Figure 2b shows a scheduling feasibility of DFG.

Techniques for improving testability applicable at the RTL level are to redesign the circuit by the addition of dedicated test structures. These structures induce an increase in the circuit surface and a degradation of speed that may violate the constraints used during synthesis. Considering testability at such a late stage in the design flow limits efficient design space exploration [17]. Furthermore the inclusion of testability at more detailed description levels is complicated by the volume of data to process. The displacement of the inclusion of testability to the behavioural level allows us to consider testability as a constraint for the synthesis (as well as surface and latency). In this way, the synthesis system seeks to generate among all solutions which respect the constraints of surface and latency, the best from the testability point of view, notably in terms of enhanced the test fault coverage. In this paper, we consider testability as a design objective alongside area and delay.

2.2 Scheduling for Improved Testability in Behavioural Synthesis

New strategies have appeared in the behavioural synthesis, to take into account testability at the same time as the surface and / or performance [10, 11, 12, 13]. In this way, testability problems can be avoided or resolved by the most appropriate decisions for the architecture definition. The scheduling technique is time constrained which minimizes the number of resources (operations) and the number of registers based on a cost function. This improves the life time of primary input and primary output variables, reduces the life times of intermediate variables and hence improves the controllability and observability [10]. The testability of the register transfer level (RTL) structure generated by this schedule is therefore improved. Particularly, in the context of on-line testability, each functional unit (FU) of a data path is tested at least once within their idle-time. A given scheduled data flow graph is utilized to estimate the number of FUs and their idle periods in which certain testing operations are scheduled [11]. Testing time is reduced by minimizing the number of types of operations assigned to each module needed to synthesize a given scheduled data flow graph (SDFG), and by creating sufficient idle time [12]. The time constrained scheduling and resource constrained scheduling take any behavioural description represented as a data flow graph as input and generate a data path composed of resources like modules, registers and multiplexers [13].

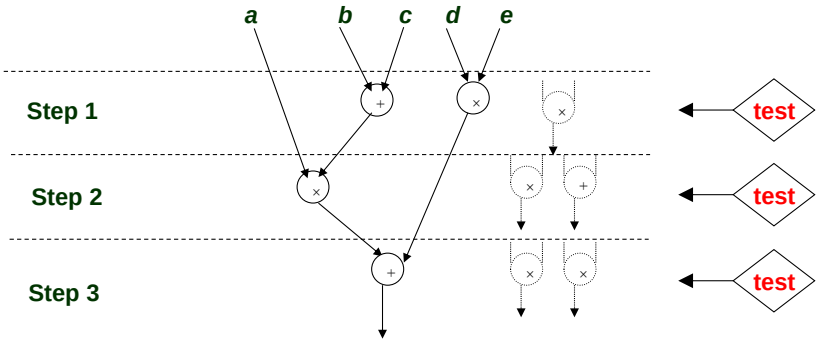

Figure 2. SDFG with idle-time operation for improving on-line testability

This improvement in testability has not required the addition of dedicated test structures and does not cause an increase in the number of hardware resources allocated. It is accomplished through an appropriate decision taken during the synthesis [4]. This example also shows that taking into account the criterion of testability during synthesis enables the generation of more testable circuits. Using this approach allows minimizing (or eliminating) the number of dedicated test structures added to the circuit and increasing the fault coverage [15].

3. EXPRESSIONS OPTIMIZATION OF DATA FLOW GRAPH

Using the associativity of the operators allows to build different versions of the same expression that are mathematically equivalent but structurally different, since for instance the position of the operands in the tree representing the expression varies. We therefore wish to count the number of structurally different trees that can be built from a DFG expression containing n distinct operands. Commutative equivalent trees are counted here only once, because they are structurally very similar. The interest of this work is to optimize the evaluation of algebraic expressions in order to give preference to those that are better suited (as containing less dependencies) with a parallel instruction execution, including scheduling and on-line testing the high level synthesis.

3.1 Counts of different commutative binary trees

Let $\oplus$ be an associative-commutative operator and X_1, X_2, ... X_n with n distinct ordered operands. We call P($\oplus$, n) the procedure for construction of all different commutative binary trees containing n operands and n-1 identical commutative operators, described by the algorithme1. The construction is done recursively by introducing at each step a new operand to all possible positions in the trees constructed in previous steps. We call "different commutative binary trees," two trees such that it is impossible to switch from one to another by application of the commutativity of the operators. Counting proposed therefore focuses on expressions such as depth of the operands in the tree is different. The notation e_{n-1} [$Y \leftarrow \oplus (Y, X_n)$] describes the substitution of the subtree Y by the subtree $\oplus (Y, X_n)$ within the tree e_{n-1}.

Algorithme 1: construction of commutative binary trees

- **Function** P($\oplus$, n) *return* E_n *the set of constructed trees*
- $\oplus$ *: a* **commutative** *operator/ n : the number of distinct ordered operands* {X_1, X_2 ,…X_n}

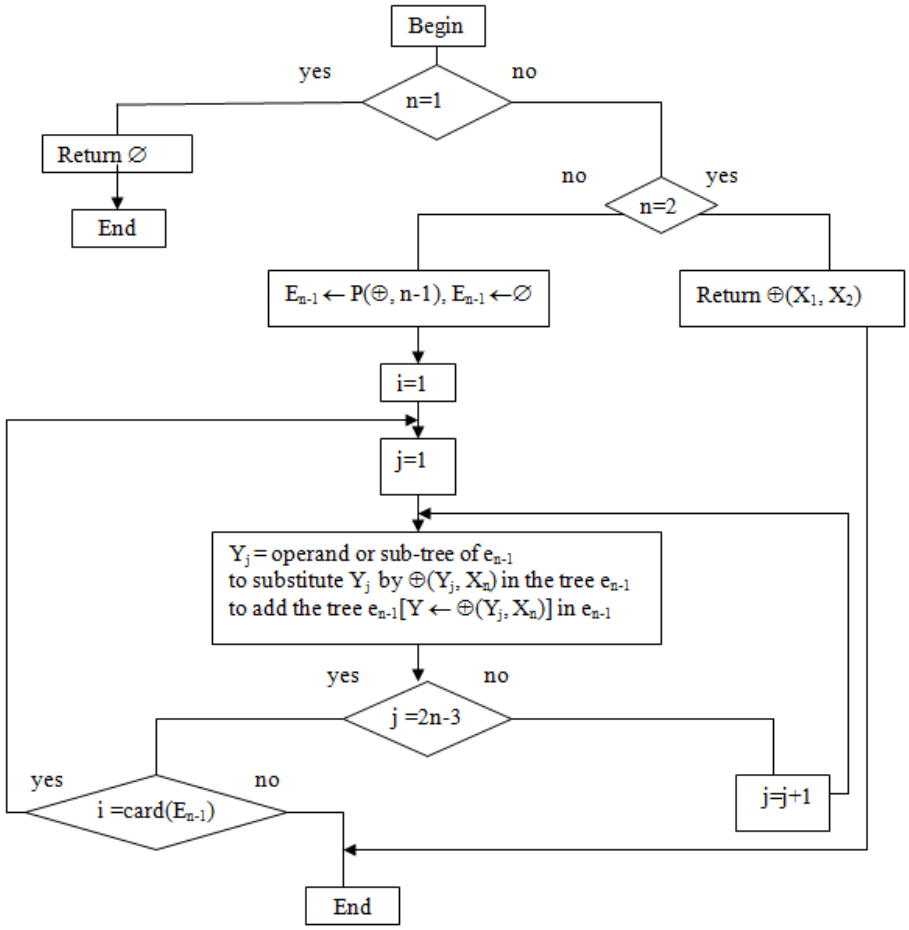

Figure 3: Flowchart representing algorithm for construction of commutative binary trees

For reasons of combinatorial explosion, it is hardly possible to browse the entire space of solutions (around n! × 2n different trees with P (⊕, n)), for each of these expressions in a program. It should also perform a static a priori evaluation of the execution time. Our approach is heuristic in nature and attempt to select from a subset of equivalent expressions, a solution with good results (in terms of evaluation time). We will seek to establish selection criteria that reflect the characteristics (number of operations, balancing, etc...) that we want, given the specific set of modern architectures and their compilers. Our goal is to exploit the algebraic properties of operators to rewrite the expressions with appropriate performance criteria. First, we are looking at ways of reducing the number of additional operations to be executed; such a transformation is mostly beneficial. Moreover, because of the parallelism of instructions, taking into account the structure of the studied expressions becomes an essential consideration.

3.3 Description of the selection criteria

We call $\mathbf{w_i}$ the weight associated with node i in the tree representing the expression. This value models the cost required to evaluate a given operation, and therefore it approaches the latency of an operation. The total weight of an expression containing n nodes (operations), noted W_E, is defined by: $W_E = \sum_{i=1}^{n} w_i$.

The total weight of an expression used to characterize the computational effort to be applied to evaluate a given expression. This value is particularly important for architecture with only a single computing unit. We note $\mathbf{d_i}$ distance or depth of a root node of the tree structure. The critical path of an expression containing n nodes, denoted C_E, is defined by: $C_E = \max d_i, 1 \leq i \leq n$.

The critical path of an expression captures the maximum length dependencies between the different operations that must be linked sequentially. The current trend is that modern processors have multiple processing units, each of which is pipelined, but without having unlimited computing capacity. The total weight and the critical path of an expression must both be considered. For this, it was defined a new measure called gravity: The gravity of an expression containing n nodes, denoted by G_E, is defined by: $G_E = \sum_{i=1}^{n} w_i \times d_i / W_E$

This measure describes the average depth of the tree operations. This "estimated cost" is a good estimate taking into account both the weight of calculations and their position in the tree. The decisions from the critical path or gravity are very close. However, gravity is used to distinguish two expressions of the same critical path but with a different number of occurrences of this critical path. Gravity is a decision criterion for effective superscalar processors with instruction-level parallelism.

3.4 Factorisation heuristic

The factorization of an expression is made to reduce the number of operations required for its evaluation. In addition to this reduction in the number of operations, we want to select, among all forms of a factorized expression, one that is best suited for execution on modern processors.

3.4.1 Factorization method

The technique is to search for a given node of the tree, all potential candidates for factorization. The algorithm proposed in Figure 4 returns a list of all factorizations found for a given node. We note in the form (*x, list of yi, z*) the factorization of an expression $E = x \times \sum_i y_i + z$. The notation $n_1[n_2 \rightarrow n_3]$ describes the substitution of node n_2 by node n_3 inside node n_1. For a node with n terms each containing m "in terms", the complexity of this algorithm is

$O(n^2m^2)$. This value is given assuming that the comparison between two sub-trees is performed in constant time O(1). Even if it's not really the case, this assumption becomes reasonable in practice through the use of a suitable data structure [18].

Function list _of_candidates(*n*)

- *Input* : n is a node
- *Output* : a list of factorizable candidates
  If (operator(n) is "-" ***then*** n=child of (n) ***end if***
  If (operator(n) is "*" ***then return*** children of (n)
  else return singleton (n)
  end if

Function <u>Factorisation(*n*)</u>

- *Input* : n is a node
- *Output* : a list of possible factorizations for this node

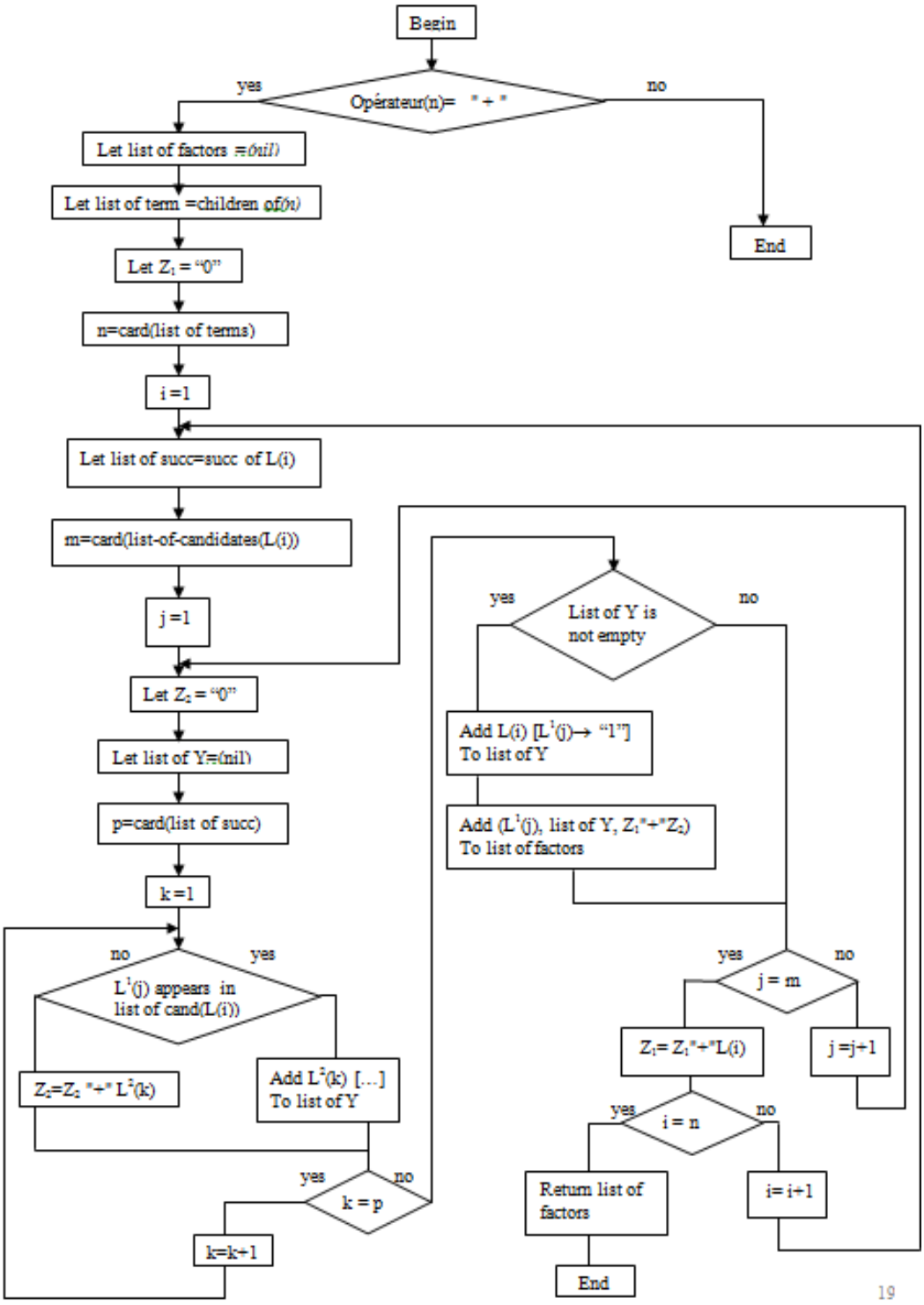

Figure 4 – Flowchart representing algorithm factorization heuristic

3.4.2 Conflict management

Several factored forms can exist for a given expression. It is necessary in this case to make a choice; sometimes, two factorizations are indeed incompatible. For example, only one factorization can both be applied to the expression: $a \times b + a \times x_1 + b \times x_2$. The result is either one: $a \times (b + x_1) + b \times x_2$, or the other: $a \times x_1 + b \times (a + x_2)$. Although the number of operations needed to evaluate this expression in both factorized forms is identical, the evaluation time can be significantly different. The respective costs of the terms involved (a, b, x_1 and x_2) influence in different effect the total cost of the expression according to the chosen factorization.

The choice between different forms of a factorized expression will be performed using as selection criteria the gravity G, and the selected factorization will be the one that minimizes the gravity so as to promote the exploitation of instruction-level parallelism. The proposed approach is based on an iterative progression through elementary transformation to an expression with "sound" properties (weight, critical path, gravity, etc.). Consider an expression containing three candidates' x_a, x_b and x_c for factorization. The expression has the following form:

$$x_a.x_b.x_c.Fabc_1 + \ldots x_a.x_b.x_c.Fabc_{nabc} + x_a.x_b.Fab_1 + \ldots x_a.x_b.Fab_{nab} + x_a.x_c.Fac_1 + \ldots x_a.x_c.Fac_{nac} + x_b.x_c.Fbc_1 + \ldots x_b.x_c.Fbc_{nbc} + x_a.Fa_1 + \ldots + x_a.Fa_{na} + x_b.Fb_1 + \ldots + x_b.Fa_{nb} + x_c.Fc_1 + \ldots + x_c.Fc_{nc} + F_1 + \ldots + F_n$$

There are six different factorized forms of this expression, the number of operations is identical for each of the six possible factorizations, but the position of terms (x_a, x_b, x_c, $Fabci$, Fab_i, Fac_i, etc.) differs according to the factorization. Figure 4 shows the structure of the tree to a factorization of x_a, x_b and x_c. The factorized expression is given by the following equation:

$$x_a.[\,x_b.(x_c.\Sigma_{i=1}\text{nabc}\ Fabc_i + \Sigma_{i=1}^{nab} Fab_i) + x_c.\Sigma_{i=1}^{nac} Fac_i + \Sigma_{i=1}^{na} Fa_i\,] + x_b.(x_c.\Sigma_{i=1}^{nbc} Fbc_i + \Sigma_{i=1}^{nb} Fb_i) + x_c.\Sigma_{i=1}^{nc} Fc_i + \Sigma_{i=1}^{n} F_i$$

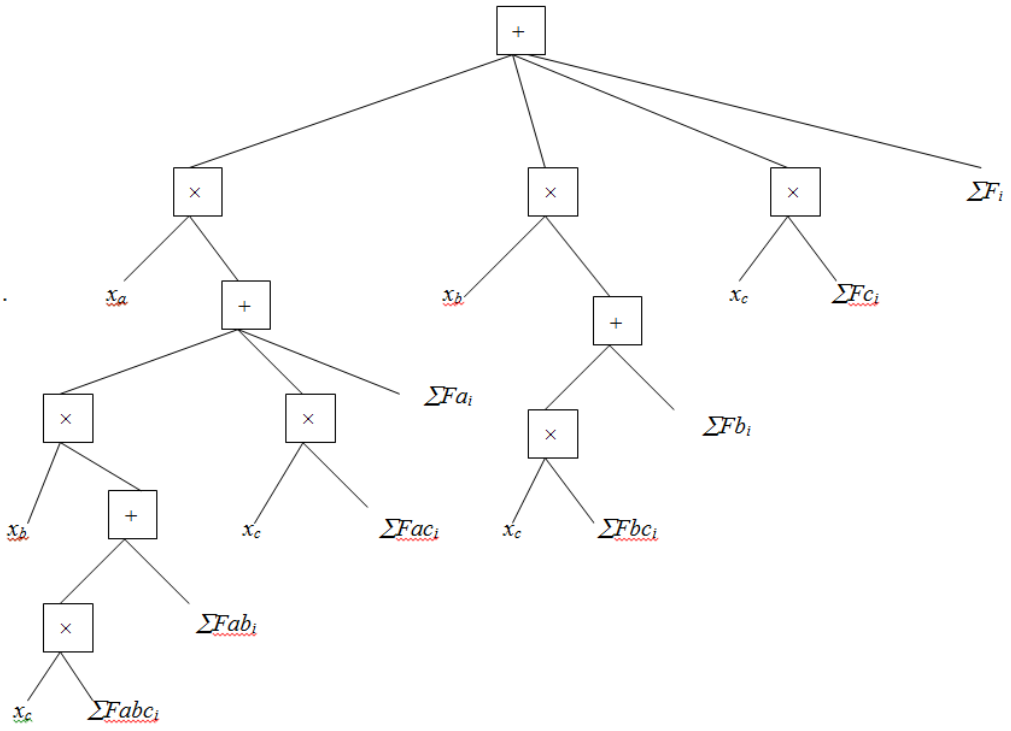

Figure 5 –Partial factorization with x_a then x_b and x_c

Tree-depth	Partial factorization with x_a	Partial factorization with x_b
1	$F_1 \ldots F_n$	$F_1 \ldots F_n$
2	x_a, $Fbc_1 \ldots Fbc_{nbc}$, $Fb_1 \ldots Fb_{nb}$, $Fc_1 \ldots Fc_{nc}$	x_b, $Fac_1 \ldots Fac_{nac}$ $Fa_1 \ldots Fa_{na}$, $Fc_1 \ldots Fc_{nc}$
3	$Fa_1 \ldots Fa_{na}$	$Fb_1 \ldots Fb_{nb}$
4	x_b,x_c, $Fabc_1 \ldots Fabc_{nabc}$, $Fab_1 \ldots Fab_{nab}$, $Fac_1 \ldots Fac_{nac}$	x_a,x_c, $Fabc_1 \ldots Fabc_{nabc}$, $Fab_1 \ldots Fab_{nab}$, $Fbc_1 \ldots Fbc_{nbc}$

Tableau 1 - Tree-depth of partial factorizations

3.5 Balancing a Tree

A transformation of the n-ary expressions in binary trees will be done because even if all binary trees that can be constructed from an n-ary tree will contain a number of identical operations, however they may have different characteristics (depth, gravity, etc.). The execution time can therefore also vary significantly.

3.5.1 Huffman coding

To select a binary tree suitable for execution on a processor with instruction-level parallelism, we will seek to intervene in the tree structure and the position of the operands. We propose to use a variation of the encoding technique introduced by David Huffman [9].

Huffman coding is an entropy encoding algorithm used for data compression: given "a set of symbols and their weights (usually proportional to probabilities)", it finds "a prefix-free binary code (a set of codewords) with minimum expected codeword length (equivalently, a tree with minimum weighted path length from the root)". The algorithm is based on the construction of a binary tree minimizing the formula $\sum_{i=1}^{n} P_i \times L_i$ with P_i the property of the symbol i L_i and its depth in the tree.

3.5.2 Construction of binary trees by a variation of Huffman coding

The similarities between this Huffman coding issue and the construction of the binary representation of an operation on n operands led us to implement a variation described by algorithm 3. The approach is similar to Huffman coding. However, the combination of two symbols in the case of coding is characterized simply by a sum of probabilities, no additional costs is introduced. However, in cost evaluation of a binary operation, the cost of the two operands and the elementary cost of the operator are involved are both involved in the global calculation.

Algorithm 3 : Construction of binary trees by a variation of the Huffman algorithm

Function Binarization (⊕, *L, C*)

- ⊕ : An *associative-commutative operator*
- *L : A* list *of n operands and L[i] the* i^ième^ *element of L*
- *C : A cost* function

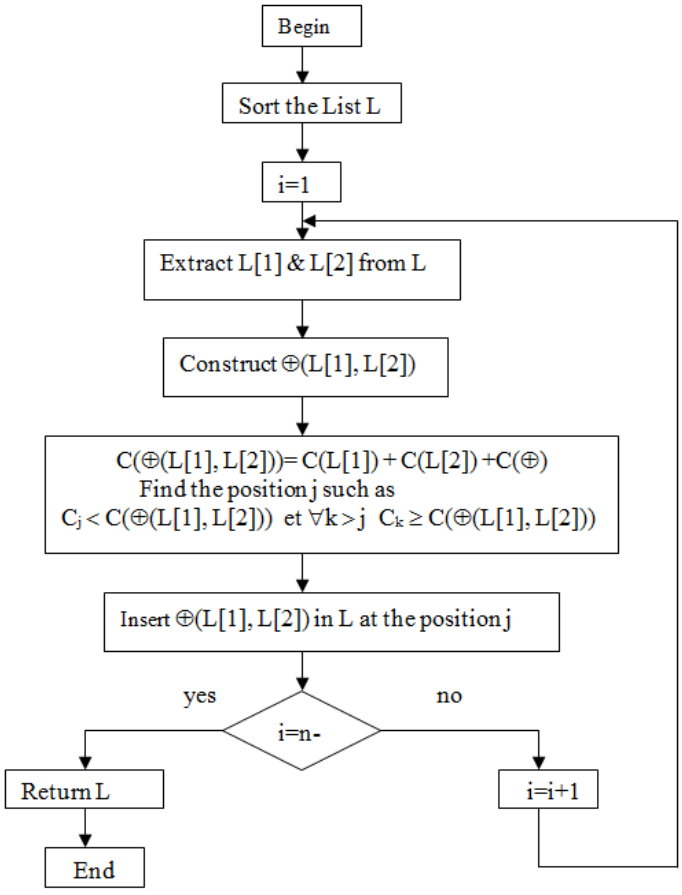

Figure 6 –Flowchart representing algorithm for construction of binary trees

Given the procedure B(n) for binarization of an operation ⊕ for n operands described by algorithm3; let c_i and d_i be respectively the cost and the depth of an operator or an operand i in the binary tree, and given the cost $c\oplus$ of the operator ⊕. By construction, B(n) minimizes the cost function: $C = \sum_{i=1}^{n} c_i \times d_i + c\oplus \times \sum_{j=1}^{n-1}$.

And consequently for $c_i = w_i$, B(n) minimizes the gravity $G_E = \sum_{i=1}^{2n-1} w_i \times d_i / W_E$.

The intervention on the balanced tree exists at the factorization level and through the construction of the binary form of the operations. The motivation of this work is to provide the expressions containing a large number of evaluable operations independently. It is usually preferable to place expensive computations at top of the tree, for example to not block the execution of lower cost calculations. The used techniques are thus

seeking to facilitate the scheduling calculations in order to improve data path scheduling scheme for easy testability [16].
Figure 7 shows the construction of a binary expression using the cost, the notion of weight defined above. The weight of the operator is set to 1 and the six operands (possibly representing any sub-tree) are weighted by the variables $w_i = c_i = i$. The creative process and the resulting expression are shown respectively in left and right of the figure.

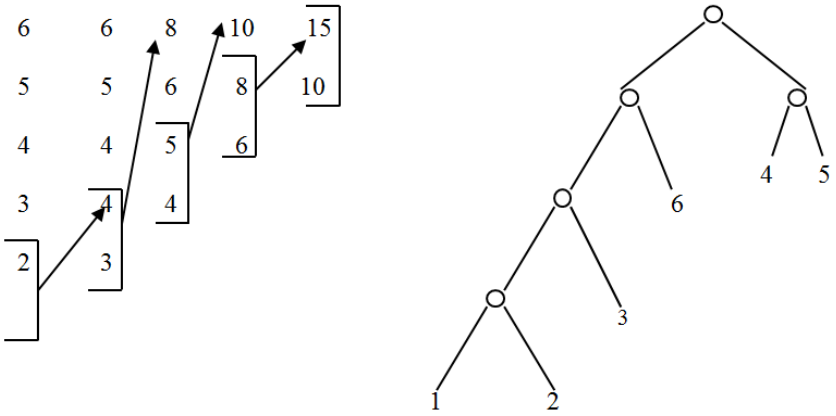

Figure 7 — An illustrative example for construction of an optimal binary tree

4. CONCLUSION

The objective of this work is to analyse the scheduling step in high-level synthesis in order to influence the testability of the synthesized circuits. Based on this study, we propose a new approach taking into account the online test a circuit at a high level of abstraction, as well as the criteria of latency and area during synthesis of architectures. For our analysis, it has been necessary both to provide an overview of the context of behavioural synthesis. This design methodology is defined as a set of refinements for compiling algorithmic descriptions of a next lower abstraction level: register-transfer level. We then place special emphasis on the influential step of scheduling and its associated data paths. Subsequently, a study of testing techniques shows the importance of taking into account the test at a high level of abstraction. Thus, we have introduced a method for data path synthesis for testability to eliminate the problems of testability as soon as possible [19].
This paper presents an efficient testability-improved data path scheduling scheme based on the algebraic properties of operators (such as commutativity, associativity or distributivity) used to optimize expression evaluation in behavioural descriptions. First experimental results have showed that the transformation of an expression in a form well suited to running on superscalar processors achieves significant gains in performance (reduction of execution time and improve reliability). We have proposed a method of selection between different equivalent expressions based on the use of criteria taking into account the number of operations, the depth of the tree or the weighted average depth of the operators in the tree. A heuristic factorization scheme characterized by an iterative mechanism of development has been introduced to reduce the number of operations of an expression. This transformation can also build the best expression compared with a selection criterion (for example reflecting the balance of the tree) determined in the high-level synthesis. An optimal algorithm based on a variation of the Huffman algorithm was used to construct binary trees, from an n-ary representation, minimizing the "estimated" cost of expression evaluation. This construction provides means of intervention in the position of each operand; however, the number of operations does not change. Finally, the two steps of our process (factorization and construction of the binary operations) build expressions with more choices not only for independent scheduling calculations, but also to improve the opportunities of testability in high level synthesis. A more recent approach uses the principle of alternate test for offline and online monitoring purposes [20].

REFERENCES

[1] Emmanuel Casseau, Bertrand Le Gal. Design of multi-mode application-specific cores based on high-level synthesis. Integration, the VLSI Journal, Volume 45, Issue 1, January 2012, Pages 9-21

[2] Coussy, P.; Gajski, D. D.; Meredith, M.; Takach, A. (2009). "An Introduction to High-Level Synthesis". IEEE Design & Test of Computers 26 (4): 8–17.

[3] Alberto A. Del Barrio, Seda Ogrenci Memik, María C. Molina, José M. Mendías, Román Hermida. A fragmentation aware High-Level Synthesis flow for low power heterogenous datapaths; Integration, the VLSI Journal, In Press, Corrected Proof, Available online 28 February 2012.

[4] Eunkyoung Jee, Junbeom Yoo, Sungdeok Cha, Doohwan Bae. A data flow-based structural testing technique for FBD programs. Information and Software Technology, Volume 51, Issue 7, July 2009, Pages 1131-1139.

[5] Ewout S. J. Martens; Georges Gielen (2008). High-level modeling and synthesis of analog integrated systems. Springer. ISBN 978-1-4020-6801-0.

[6] Dimitris Gizopoulos. Low-cost, on-line self-testing of processor cores based on embedded software routines.Microelectronics Journal, Volume 35, Issue 5, May 2004, Pages 443-449.

[7] Michael Fingeroff. High-Level Synthesis Blue Book. Xlibris Corporation. ISBN 978-1-4500-9724-6, 2010.

[8] Christoforos N. Hadjicostis. Periodic and non-concurrent error detection and identification in one-hot encoded FSMs. Automatica, Volume 40, Issue 10, October 2004, Pages 1665-1676.

[9] David Huffman, "A method for the construction of minimum-redundancy codes" In Proceedings of the I.R.E, pages 1098-1101, September 1952.

[10] A.A. Ismaeel, R. Mathew, R. Bhatnagar. Scheduling and variable binding for improved testability in high level synthesis. Computers & Electrical Engineering, Volume 24, Issue 6, November 1998, Pages 441-461.

[11] A.A. Ismaeel, R. Bhatnagar, R. Mathew. Modification of scheduled data flow graph for on-line testability. Microelectronics Reliability, Volume 39, Issue 10, October 1999, Pages 1473-1484.

[12] A.A. Ismaeel, R. Bhatnagar, R. Mathew. Module allocation with idle-time utilization for on-line testability.

Microelectronics Reliability, Volume 41, Issue 2, February 2001, Pages 323-332.
[13] A.A. Ismaeel, R. Bhatnagar, R. Mathew. On-line testable data path synthesis for minimizing testing time. Microelectronics Reliability, Volume 42, Issue 3, March 2002, Pages 437-453.
[14] Saeed Safari, Amir Hossein Jahangir, Hadi Esmaeilzadeh. A parameterized graph-based framework for high-level test synthesis. Integration, the VLSI Journal, Volume 39, Issue 4, July 2006, Pages 363-381.
[15] Emmanuel Simeu. Optimal Detector Design for On-line Testing of Linear Analog Systems. VLSI Design Volume 11 (2000), Issue 1, Pages 59-74 doi:10.1155/2000/92954.
[16] Emmanuel Simeu, Ahmad Abdelhay, Mohammad A. Naal: Robust Self Concurrent Test of Linear Digital Systems. Asian Test Symposium 2001:293-29.
[17] M. Zwolinski, M.S. Gaur. Integrating testability with design space exploration. Microelectronics Reliability, Volume 43, Issue 5, May 2003, Pages 685-69.
[18] Julien Zory, Fabien Coelho: Using Algebraic Transformations to Optimize Expression Evaluation in Scientific Codes. IEEE PACT 1998: 376-384.
[19] Bernard Kamsu-Foguem. Knowledge-based support in Non-Destructive Testing for health monitoring of aircraft structures. Advanced Engineering Informatics, Volume 26, Issue 4, October 2012, Pages 859-869.
[20] Emmanuel Simeu, Hoang Nam Nguyen, Philippe Cauvet, Salvador Mir, Libor Rufer, Rafik Khereddine: Using Signal Envelope Detection for Online and Offline RF MEMS Switch Testing. VLSI Design 2008 (2008).

BIOGRAPHIES

Bernard Kamsu-Foguem is currently a tenured Associate Professor at the National Engineering School of Tarbes (ENIT) of National Polytechnic Institute of Toulouse (INPT) and leads his research activities in the Production Engineering Laboratory (LGP) of ENIT-INPT, a research entity (EA 1905) of the University of Toulouse. He has a Master's in Operational Research, Combinatorics and Optimisation (2000) from National Polytechnic Institute of Grenoble, and a PhD in Computer Science and Automatic (2004) from the University of Montpellier 2.

His current interests are in Knowledge Discovery and Data Mining, Knowledge Representation, Formal Visual Reasoning, Ontology-based Semantic Analysis, Knowledge Exploitation for Collaboration, Decision Support Systems and Intelligent Systems. Application domains include Continuous Improvement process, Industrial Maintenance management, Health Information Systems and Alternative Medical Systems.

He has authored or co-authored a number of papers in the international scientific journals such as Expert Systems with Applications, Decision Support Systems, Engineering Applications of Artificial Intelligence, Computers in Industry, Advanced Engineering Informatics, Annual Reviews in Control, International Journal of Production Research and Information Systems Frontiers.

He is a reviewer for a large number of international scientific journals such as Engineering Applications of Artificial Intelligence, Concurrent Engineering: Research and Applications, Artificial Intelligence Research, Canadian Journal of Administrative Sciences, International Journal of Computer Engineering Research, Knowledge Management Research & Practice, Journal of Intelligent Manufacturing, International Journal of Production Research, Interacting with Computers, and Knowledge-Based Systems.

Dr. B. Kamsu-Foguem was recently awarded two prizes (Best Paper Award in 2009 and 2011 from the SKIMA - IEEE conferences) and one audience distinction (Most Downloaded Engineering Applications of Artificial Intelligence Article from SciVerse ScienceDirect in 2012) for his research topics in continuous improvement, knowledge reasoning and maintenance management. He is interested in the international network and collaboration with other institutions and researchers related to research projects, course development and delivery.

Emmanuel Simeu received Master of Science and Ph.D. in automatic control and theory of systems from National Polytechnic Institute of Grenoble in 1988 and 1992 respectively. He has been researcher CNET from 1989 to 1992 and researcher in Automatic Control Laboratory of Grenoble (LAG) from 1988 to 1995. Since 1995 Emmanuel Simeu is associate professor in Electrical Engineering and Automatic Control in Polytechnic Institute of Joseph Fourier University of Grenoble.

He also is researcher in Reliable Mixed Signal systems (RMS) group of TIMA Laboratory and received the HDR (accreditation to supervise research) degree in Physics from Joseph Fourier University of Grenoble. His research interests include complex system modelling, reliability of heterogeneous integrated systems, diagnosis and control of analogue, digital and mixed signal embedded systems.

Emmanuel SIMEU is author and co-author of over a hundred publications of papers in international journal and conference proceedings. He has supervised twelve Ph.D. theses (including five in progress).

He is currently Director of Computer Integrated Manufacturing (CIM) research and development platform of AIP-PRIMECA-DS. He is actually the chair leader of Afrisciences Journal.