VLSI DESIGN
1998, Vol. 7, No. 4, pp. 321–336
Reprints available directly from the publisher
Photocopying permitted by license only

A High Level Synthesis System for VLSI Image Processing Applications

FRANÇOIS S. VERDIER[a,*] and BERTRAND ZAVIDOVIQUE[b]

[a]*ETCA Système de Perception Laboratory, 16 bis Avenue Prieur de la Côte d'Or, F-94114 Arceuil France;*
[b]*Institut d'Electronique Fondamentale, University of Paris-XI, F-91405 Orsay France*

(Received 19 August 1994; In final form 10 March 1995)

We present a VLSI synthesis environment dedicated to the design of image processing architectures. The environment includes a "front-end" data-flow emulator for validation of the algorithms and the RTL-synthesis system called ALPHA. The latter implements a stochastic search in the design space and produces efficient solutions considering the "restricted" domain of concerned applications. Two simulated Annealing (SA) algorithms run in sequence for data-path synthesis (scheduling and module selection) and then for control synthesis and data-path completion (binding). An interesting feature of the first optimization is the use of the data-flow graph regularity to predict the control influence in terms of the future design. A few designs have already been compiled under this environment including a default detector presented here.

Keywords: High-level synthesis, image processing automata, functional decomposition, embedded systems, VLSI architectures, control optimization

1. INTRODUCTION

Today's demand for both high performance and severely constrained machine vision functions leads to an ever increasing diversification of algorithms while design cycles get shorter. Although the computing power of general purpose micro-processors and DSPs is dramatically increasing accordingly, it still is surprisingly difficult to find a good enough (in terms of performance, real-time execution, memory sizes...) match between a complex Image Processing algorithm and any such traditional architectural target. This is mainly due to the topological diversity of both computing devices and behavioral descriptions. An other reason is a lack of a uniform description formalism for Image Processing applications which significantly slows down the specification, validation and integration phases. When a description exists, it remains strongly problem oriented or target architecture dependent. The need for a single formalism describing concurently what a function does (the behavioral description) and how it can be

*Corresponding author.

performed (the corresponding structural description) remains a major challenge.

Considering the system level, both technological and economical impacts of efficient High Level Synthesis systems do not need to proved once again. *Simulating*, taking into account the whole complexity of VLSI systems, *handling* higher and higher integration needs and *proving* the functional equivalence between the initial specification and the specific synthesized solution are the inevitable functionalities of tomorrow's VLSI design systems.

A global approach to the automatic synthesis of complex Image Processing systems appears then necessary. The automatic mapping of algorithms onto specific architecture targets is the majority unsolved yet problem.

We advocate for a "functional approach" to describing both algorithms and architectures through a unique formalism. The "functional decomposition" of vision tasks has been studied in our lab for approximately a decade and has led to the design of three complete environments: one for algorithmic design based on analogies [1], the second for hardware automata emulation based on ASICS assembling and inituitive programming [2] and the third, an intermediate version, to be used here [3, 4]. It is a specialized highly parallel computer coupled to the high level synthesis system. The architecture called the Data Flow Functional Computer (DFFC) is dedicated to the real time emulation of Image Processing applications expressed in a Functional Programming language and represented by their directed Data Flow Graphs (DFG). The synthesis system (ALPHA) consists in two stochastic based optimization algorithms (scheduling and binding) and performs the Register Transfer Level (RTL) synthesis of the validation algorithm from the DFG representation.

The associated methodology aims at rapidly and efficiently design embedded machine vision systems satisfying stringent environment-specific constraints. It relies on:

- the decomposition of an algorithm into a set of well known functional primitives,
- the study of its behavior on the Functional Computer,
- the automatic integrated derivation onto hardware devices.

Within this methodology the phases of specification, validation and hardware implementation are supported by a single formalism: the *functional description*.

This paper is organized as follows: the second section describes this so-called functional decomposition of Image Processing algorithms and its use for both algorithm and architecture descriptions, Section 3 presents the Functional Computer and its programming environment. In Section 4 we present in detail our High Level Synthesis system ALPHA, the scheduling and binding algorithms that are similarly based on a Simulated Annealing (SA) optimization heuristic including VHDL interface. Some results are shown in Section 5. The example of synthesized circuit here is default detector.

2. FUNCTIONAL DECOMPOSITION

Research on algorithms dedicated to Image Processing and experience in implementing them on computing hardware have led us to propose a new methodology when designing Image Processing systems. This methodology is based on decomposing a complex IP task into smaller communicating functions. Within this methodology, extracting a semantically complex feature from a collection of pixels can be viewed as low level functions extracting elementary information from the image, communicating this information to higher semantic level functions and so on until the final analysis is performed. All these elementary functions can work independently or cooperatively. This method is known as the *functional decomposition*. Three basic catogories of functions appear in this decomposition:

1. Low level processing covers the elementary analysis of an image. All the processed data are pixels (binary, gray levels, "colors") and only

local neighborhoods are concerned. Low level functions extract basic features (segmentation functions).
2. Medium level functions take these feature elements as their input (gradients, interest points) and generate symbolic results (regions, surfaces, lines).
3. High level processing aims at completing a high semantic analysis of the image by matching complex data (rule based recognition, manipulation of complex object graphs).

This knowledge partioning leads to the *a priori* definition of operations (the primitives) assigned to the extraction of only a given part of the global information. Noise reduction, edge detection, region growing, pattern matching are some examples of such primitive sets. Not all primitives of a set have similar data movements, similar needs in terms of memory and communications: for instance conventional basic edge detection might be a mere filtering, adaptative or not, asking for local computations in parallel, and then some global optimization for edge linking and description, like a vote or some relaxation. Edge detection might be as well a tree search-like a branch-and-bound or some dynamic programming to edge finding through edge following based on local contrast.

Then, mapping a complex algorithm onto a specific computing topology is far more difficult. One generally tries to find the best trade-off between space (amount of available computing resources) and time (amount of time needed to compute the whole process on this architecture). This is mainly due to the symbolic gap between an algorithmic description (rather heterogeneous) and the available architecture description (often monolythic).

By designing a new architecture, based on a new formalism for its description, we have found a way for an easy mapping of a functional behavioral description onto computing devices. The drawback is that not all operations are implementable in a straightforward manner anymore: some functions have to be approximated or redesigned.

Nevertheless, advantages of such a functional description are numerous:

- Intuitive programming: this kind of a representation is more natural than procedural languages and other C-like descriptions,
- Hierarchical specification: complex Image Processing functions can be developed using a top-down methodology starting by specifying the global process without going down to the precise behavior of low-level processes,
- Hierarchical integration: spatial and temporal symmetries can be easily shown from the specification and results are separately integrated as macro functions. In doing so, a hardware module database is expanded with more semantically complex macro blocks. An example of such hierarchical synthesis is shown with the default detector example in Section 5.

3. EMULATION: THE FUNCTIONAL COMPUTER

3.1. Architecture

Part of our method, namely the validation of an algorithmic solution for a given IP problem, is done prior by implementing it on a dedicated Data-Flow Computer [3]. This Functional Computer is made of a 3D network of 1024 custom data-flow processors (the functional Processors). Every processor is built around a Configurable Data-Path with three inputs and three outputs connected to six Input/Output Ports through a crossbar routing cell. This structure allows the data-flow processor to be mesh-connected to six neighbors in a 3D fashion. The algorithmic capabilities of these processors are typically medium-level Image Processing applications, based on a multiply-ALU pipeline, counting and memory accesses that enable for instance histogramming as a basic instruction.

The computer also contains several Input/Output high and low bandwidth channels for both B & W and color image sources. These channels allow a realistic dynamic evaluation of the algorithms.

In addition to the execution of binary and arithmetic operations, the elementary Functional Processor can be configured as a routing element allowing local connections between operative elements. Figure 1 gives an overview of the Functional Computer structure.

On top of the low-level Functional Processors network (the Field Programmable Operator Array), up to 36 2D-mesh-connected INMOS Transputers allow some higher level functions to be implemented. Due to the Flow execution mode (data driven) of the low-level network, special synchronization functions are available for interfacing both networks.

The algorithms are mapped onto this computing structure using their Data-Flow Graph representations. A large graph as well as several medium graphs may be implemented and executed independently.

3.2 Programming Environment

The first step for programming the Functional Computer is to describe the algorithm with a functional language (FP – Functional Programming language). This input language is based on the FP language developed in [5] and is outlined later in this paper. A compiler (Functional Programming language Compiler : FPC) translates the input description into a data-flow graph. The data-flow graph represents the algorithm in terms of abstract primitives (binary or arithmetic operations, more complex macros, Image Processing functions ...).

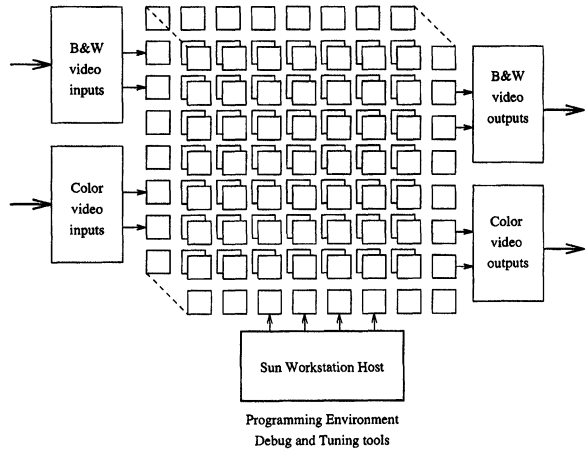

FIGURE 1 Functional Computer Architecture.

This graph is then translated into a more hardware-specific graph formed by a set of interconnected processors (functional units, hardware macros and FIFO devices for load balancing). This is done in matching every node with a known function in the database. This database includes the libraries of lower level hardware implementation, one for each primitive. Three types of libraries are available:

- The low level processor library contains 150 functions which can be mapped onto a single Functioanl Processor.
- The macro library contains 50 functions mapped onto a set of processors already interconnected. This library can be extended by the programmer by using a graphical backend interface with the computer.
- The C-language library contains about 10 predefined functions for programming the high-level network (Transputers). This library can be easily extended using a classical C-programming environment.

The *hard ware graph* is finally mapped onto the cubic network and is ready for execution and performance evaluation on the fly. The routing facilities of the elementary data-flow processor as well as several communication ports allow a limited loss in density when mapping an algorithm. One can easily obtain a density in the range of 70% (computing devices/total routing nodes). The ultimate automated placing and routing tool is still under development: a first version is available for graphs with less than a thousand processors. Figure 2 shows these different phases.

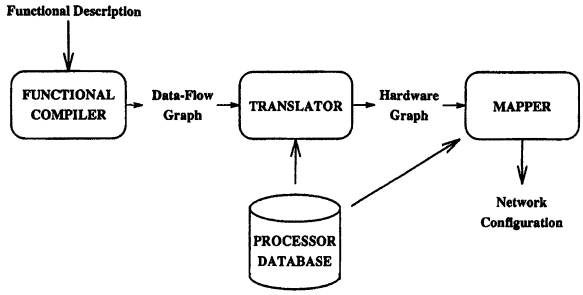

FIGURE 2 Functional Computer programming environment.

As an example of such a mapping of a complex Image Processing function onto this architecture, the whole default detector algorithm (detailed in Section 5) occupies about 290 data-flow processors.

4. SYNTHESIS: THE ALPHA SYSTEM

We present in this section the overall synthesis system called ALPHA and shown in Figure 3.

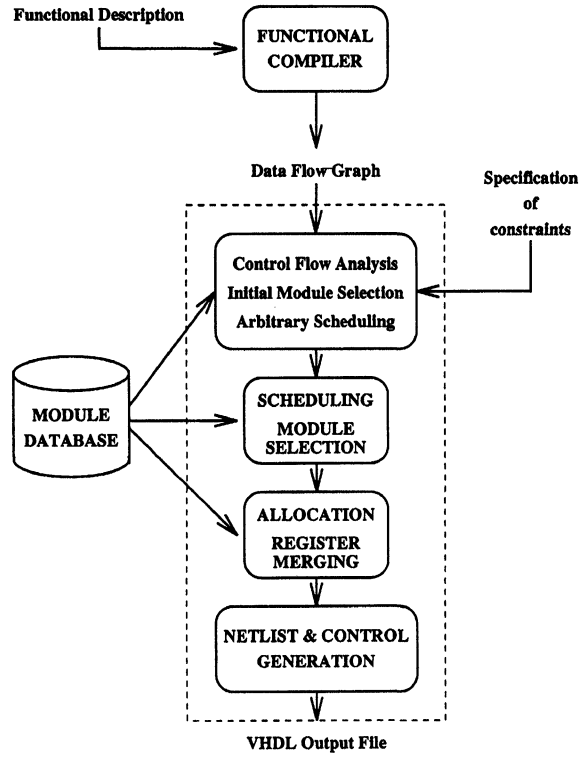

FIGURE 3 Overview of the ALPHA synthesis system.

4.1 Input Specification

As detailed in Section 2, the input representation is a Data Flow Graph translated from a functional specification. The FP compiler generates the corresponding Data flow Graph from the functional description. It allows hierarchical specifications (programs can be viewed as functions which are map products of functions). The syntax of the language used to describe algorithms is very simple and can be formalized as a triple $\langle O, F, P \rangle$ where:

- O is a collection of basic objects (data) which can be typed (binary data, 8 bits, signed integer...) and structured (sequences, lines ...).
- F is a set of functional forms (composition and construction forms) allowing combination of functions in order to build the program.
- P is a set of predefined functions representing the primitives which can be applied to O. Functions with multiple inputs and/or outputs can be defined.

By using this formalism there is a one-to-one correspondence between a functional description and a data-flow graph with explicit function parallelism and no extra dependencies. As discussed in section 2, the hierarchical flavour naturally supported by the functional description might lead to a merge of strongly dependent primitives allowing easy handle of such macro-functions. In addition, such a macro-function might have its own semantic (a contrast enhancer, a complete edge extractor,...) and therefore its own data type (a list of edge points). The typeset of the functional objects can then be extended with more semantically complex types up to the Image Processing Data Type set (interest points, movement vectors, regions, ...).

Figure 4 shows an example of a functional program and its corresponding data-flow graph (see Section 5 for more details).

4.2 Internal Representation

The first of the synthesis consists in reading the input data flow graph specification, of propagating

```
// FP source code of main process

MAIN [
INPUT Image:PIXEL;
INPUT Thresh:PIXEL;
OUTPUT Res:PIXEL;
]
def dx = sub . [Image, Pixel_delay . Image];
def dy = sub . [Image, Trame_delay . Image];

def angle = xor . [sgn . dx, sgn . dy];
def adx = abs . dx;
def ady = abs . dy;
def m = max . [adx, ady];

def tg1 = atg_lut . [adx, ady];
def tg = norm . [tg1, angle];

def edge = thr . [Thresh, m];
def direction = select . [tg, edge];

def d0 = dir_macro(dir = 0) . [direction, edge];
def d1 = dir_macro(dir = 1) . [direction, edge];
def d2 = dir_macro(dir = 2) . [direction, edge];
def d3 = dir_macro(dir = 3) . [direction, edge];

def Res = max ! [d0, d1, d2, d3];
END
```

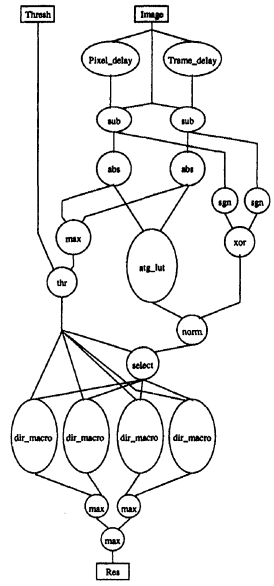

FIGURE 4 Correspondence between the functional description of the default detector algorithm and its data-flow graph representation .

data types through the graph and creating the control flow graph. These steps allow building the internal representation : the **Control Data Flow Graph**.

4.2.1 Data Type Propagation

As detailed in 4.1, each data flow object in the functional description (and the corresponding input and output vertices in the data flow graph) is typed. In order to select properly implementing candidates for each functional vertex in the graph, one must know the type of data flowing through the vertices. This is done by using a graph-walk-though algorithm solving the output data type for each node. Starting from all the input nodes (that are fully specified), the algorithm propagates the data flow type through the garph. At this time, three cases may occur:

- The functional primitive has a parameter specifying its output data type: **MULT(type = signed integer). [X1, X2]**. In this example, the implementing candidate for vertex MULT needs a signed-16 bit output while its input port widths are given by the types of data flows X1 and X2. The parameter is given by the compilation of the initial functional description.
- The types of each input flows are identical and no type parameter is given. This is the default case and the output type of the implementing candidate is the same as of the input flows.
- The types of the input flows differ and no type parameter is given. The output data type computation is quite complicated in this case (depending upon the operation among other things). One particular solution has been implemented resolving the operation type by computing **MAX (type(X1), type (X2))** whenever **type** ( ) is the bit-width of the flows. When processing more complex data types (for instance Image Processing structured types), this case has to be extended with other particular operatioons on type values.

4.2.2 Creating the Control Flow Graph

Some predefined primitives in the language specify a particular control sequencing. The conditional execution primitives are the IF and CASE constructs. These functions have the following syntax:

- DEF y = IF. [cond, X1, X2] where *cond* is the data flow to be tested and X1, X2 the mutually exclusive data flows. X1 is executed is the result of *cond* is True (equal 0) and X2 otherwise.
- DEF y = CASE· [cond, X1, X2, X3, . . . , Xn]. The nth exclusive flow is executed if the result of *cond* is n.

Loops are not currently implemented since they are very difficult to express in a functional formalism and to validate in real time on a data flow architecture. Nevertheless, iteration functions can be expressed in the functional language:

- Recursive functions: WHILE·[PRED, FUNC]·X is defined as X if PRED (X)= *False* and WHILE. [PRED, FUNC]. FUNC(X) otherwise.

Unlike in most synthesis systems where conditional branches are represented by adding some special nodes in the data-flow graph (*Fork* and *Join* nodes in [6], *Distribute* and *Join* nodes in USC ADM system [7, 8], *Select* nodes in the CMU-System Architect's Workbench [9] and recently *Conditional Begin* and *Conditional End* nodes in [10]) during the control flow preprocessing, all such nodes getting no truly functional implementation are removed and replaced by conditional branches in the control graph. This control graph is the graphic outline of a finite state machine controlling the architecture. The two graphs are then merged into a single Control Data Flow Graph (CDFG) which is the main representation of the architecture to synthesize. Figure 5 shows a sample FP program with conditionals and its CDFG representation.

This structure is a mixed data and control flow graph with two types of dependencies:

- Data dependencies represent precedence relationship between functional primitives as they are specified in the functional description. An edge between two operations indicates a data trasfer between two hardware modules. There are no extra inter-operation dependencies because of the absence of arbitrary variables in the program.
- Control dependencies represent the sequential behavior of the architecture. If several functional nodes belong to a single control, all these operations must be executed before reaching the next control state node.

```
MAIN [
video input X1:pixel;
video input X2:pixel;
video output y:pixel;
]

def a = geq . [X1, X2];
def b = add . [X1, X2];
def c1 = add . [b, X2];
def c2 = add . [b, X1];
def y = if . [a, c1, c2];
end
```

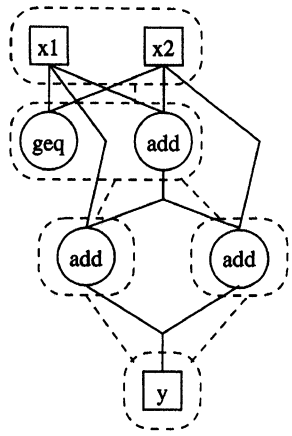

FIGURE 5 The CDFG representation (dashed lines and circles represent the control graph).

Parallel and sequential constraints are fully described by these two kinds of relationship.

In addition to the control flow analysis, the first step of the synthesis consists in arbitrarily scheduling the graph (assigning a control step to every operation in the graph). The default length of the machine cycle is the greater operation delay. This means that all nodes can be placed within a single cycle (or multiple nodes might be chained within a cycle). However, a machine cycle length can be selected regarding to the external synchronization constraints. In this case (and whenever a slow module is selected for implementing a single-cycled node), multicycling nodes might appear (hardware modules taking more than one cycle for execution) and would be processed identically.

The arbitrary scheduling is an "As Soon As Possible" (ASAP) construction with all input and output nodes (which are considered as functional nodes) scheduled in the same cycles (the first and last respectively). The initial ASAP scheduling is not critical due to the optimization method. The convergence proof of the Simulated Annealing process being assumed by a random initial solution constructed after a full perturbation of all nodes and a slow enough cooling schedule [11].

4.3 Scheduling Algorithm

Once an arbitrary solution has been given, a first optimization procedure is run to find the optimal scheduling of the data-flow graph on a synchronous machine. Module selection is achieved simultaneously. This is done by using a Simulated

Annealing based stochastic search. Let us describe in detail the technique that already appears in [12] and [13] .

The main optimization phase consists in applying a randomly selected transformation on the data-flow graph representing the architecture. The simulated annealing process iterates these transformations until a "good" solution is reached (Fig. 6). There are three kinds of transformations:

- **Move-Up**: this transformation applies a movement on a randomly selected operation node in the graph. The node is displaced to an earlier cycle if it does not break its data dependencies. In so doing, the functional equivalence of the transformed solution is maintained.
- **Move-Down**: this transformation selects a later cycle for the execution of a randomly selected node.
- **Move-Select**: a new hardware module is picked up from the database for a node to be implemented. The name of the functional node is matched against the list of potential operations of this module, completing the selection, It allows an optimal use of complex functional units as ALUs.

Currently, optimization transformations are only applied on the implementation **(Move-Select)** or sequential (**Move-Up** and **Move-Down**) views of the operations. No behavioral transformations are involved. Nevertheless, our system is soon to be completed with some optimization bound to the associativity feature of operations as well as to the delay operator moving through the graph like in [14]. These transformations have been shown to be efficient by making the explored design space larger.

```
Generic Simulated Annealing Algorithm
  Begin
    S := Initial Solution
    T := Initial Temperature
    While (stopping criterion is not satisfied) do
     Begin
       While (not yet in equilibrium) do
        Begin
          S' := some random neighboring solution of S
          dE := cost(S') - cost(S)
          Prob := min(1,exp(-dE/T))
          if random(0,1) < Prob then S := S'
        End
       Update T
     End
    Output Best Solution S
  End
```

FIGURE 6 Overview of the generic Simulated Annealing Algorithm.

The acceptance of a particular transformation is based upon an energy variation. Energy reflects quality of a solution: it is a function of several terms to minimize:

- The overall surface of the data-path under synthesis, computed from the areas of selected modules.
- The total number of machine cycles.
- The number of states in the control graph.
- A term called *regularity* translating an *a priori* complexity of the control part.

The fist two terms constitute the cost function already presented and discussed in [13] as refering to the datapath synthesis. The last two terms have been added to support control optimization. The last one is discussed below as a fundamental parameter in the control optimization process.

4.3.1 Regularity Optimization

A first efficient use of the regularity notion appears in [15]. The regularity of a data-flow graph has been also considered successfully in [16]. In that cases, the main goal is to decompose and partition FSM's ([15]) or complex designs ([16]). In addition, Rao uses his regularity notion in order to take into account data-path and control trade-offs in [17]. However, its method is still based on system clustering. In our methodology, the regularity measure has been added to the cost function and has been shown to reflect the complexity of the control part in terms of the number of control signals [18]. This is a measure of the diversity of lengths and types of data transfers within the data-flow graph. We now give some

definitions and propositions formulating the regularity of a data-flow graph.

DEFINITION 1 A node of the CDFG represents a boolean or arithmetic operation. Such a node has as many links as the number of operands and outputs in the underlying operation. The node type is identical to the one of the operation.

While scheduling, each node is temporarily assigned a specific machine cycle or step.

DEFINITION 2 An edge in the CDFG represents an interconnection facility (bus, multiplexors ...) allowing data transfers between different operatores. All edges in a data-flow graph are directed. The length of the edge between two nodes in the CDFG is defined as the number of machine cycles between these nodes.

DEFINITION 3 A motif in the graph is an item representing both the length of an edge and its associated couple of nodes (source and destination). Since our data path model is a register-mux-operator model, every motif corresponds to a data transfer between two operations (see an example Fig. 7).

Motifs are sorted out according to their type and value. The type of a motif is given by the type of the source and destination nodes for its associated edge. Its value is the internode distance – its length in terms of the number of machine cycles.

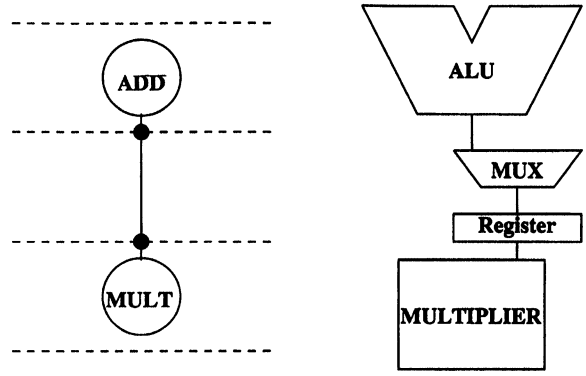

FIGURE 7 A motif and its corresponding interconnect structure.

Our method consists in computing the regularity of a scheduled graph from a statistical measure of the diversity of its motifs.

Conjecture 1 The more regular a scheduled graph (in terms of its data transfers and the number of operations in a given machine-cycle), the simpler its associated control (in terms of area, number of control signals) (Fig. 8).

One can find an illustration of this conjecture with systolic architectures where the control part is reduced to a mere clock distribution. We shall see that in such a case the regularity is infinite. Examples in Figure 8 depict the impact of increased regularity on the control complexity and on the data path implementation as well. The increase in interconnection complexity (bus, registers, multiplexors) in the irregular example of Figure 8 leads to a raise of control complexity (more control signals) and area.

We can notice that:

Remark 1 As mentioned in [19], the configuration with the lowest global cost is not necessarily

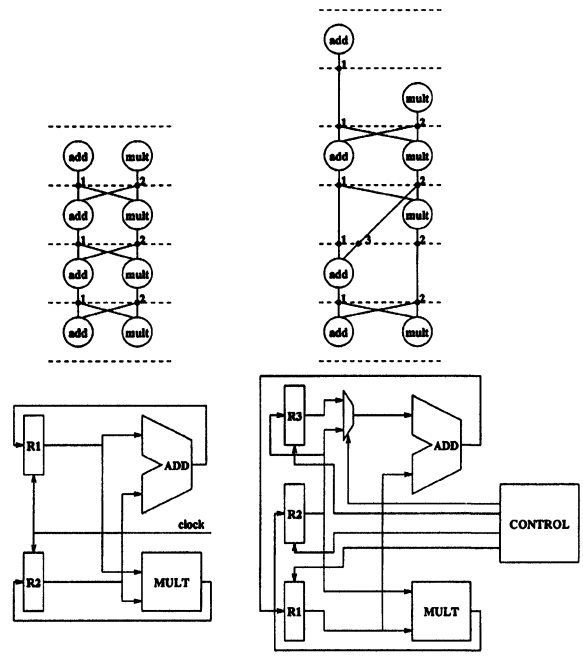

FIGURE 8 Regular (left) and irregular (right) graphs with their implementation.

the one with the minimum number of registers. Therefore, in the scheduling process, the control complexity is not a smooth function of the number of registers. Moreover, the number of multiplexors seems very difficult to state prior to the binding process (before assigning registers and operators).

Remark 2 The potential regularity of a scheduled graph depends on the topology or nature of the algorithm to be implemented. Thus, the regularity to be obtained with a particular graph is bounded by some fixed value.

How can the regularity of a scheduled graph be quantified? The solution we have selected consists in decomposing a graph into a set of motifs, and performing a statistical analysis of these motifs.

The advantages of such an approach as follows:

1. A motif gives an indication about the number of data transfers between two nodes – thus between two corresponding operators (data transfers are the most costly in terms of needed control signals).
2. Statistics inherently compress data; whence easier handling of a large amount of characteristics needed in these processes and some computation speed up.

These "motifs" are typed (definition 3) and the types are finite. Let T be the number of different operations, one gets T^2 types of motifs that statistical features have to be expressed on. Moreover the total number of edges in the graph is well known and constant (a data transfer never appears or disappears during the scheduling process) so classical statistical measures such as mean and standard deviation are adequate to evaluate the graph regularity. For each set of motif types, we compute a standard deviation of the motif values that relates to the global regularity of the graph. Below is a mathematical formulation of the regularity computation:

Let *CDFG* be the directed hierarchical data-flow graph:

$$CDFG = (X, U) \tag{1}$$

with:

$$X = \{\text{node}_1, \text{node}_2, \ldots, \text{node}_N\} \quad \text{and}$$
$$U = \{\text{edge}_1, \text{edge}_2, \ldots, \text{edge}_M\}$$

Among other features, each data-flow node is typed and *type* (node_i) stands for the module instance implementing node $node_i$ if an edge $edge_l$ exists between $node_i$ and $node_j$ then:

$$\text{orig}(\text{edge}_l) = \text{node}_i \text{ and } \text{dest}(\text{edge}_l) = \text{node}_j$$

and

$$\text{type}(\text{edge}_l) = (\text{type}(\text{node}_i),\ \text{type}(\text{node}_j)) \tag{2}$$

we also have the following sets:

$$T_{t_i} = \{\text{node}_i | \text{type}(\text{node}_i) = t_i\}$$
$$T_{t_i t_j} = \{\text{edge}_i | \text{type}(\text{edge}_i) = (t_i,\ t_j)\}$$

Their cardinals are respectively denoted by N_{t_i} and $N_{t_i t_j}$. We will now use the following notation: $L^i_{t_s t_d}$ is the length of the motif $M^i_{t_s t_d}$, where t_s is the source type and t_d the destination type.

The motifs are defined by:

$$M^i_{t_s t_d} = \text{edge}_i \text{ with } \text{type}(\text{edge}_i) = (t_s\, t_d) \tag{3}$$

and

$$L^i_{t_s t_d} = \text{length}\,(M^i_{t_s t_d}) \tag{4}$$

- Average of motif lengths of type $t_s t_d$:

$$\mu_{t_s t_d} = \frac{\sum_{i=1}^{N_{t_s t_d}} L^i_{t_s t_d}}{N_{t_s t_d}}$$

- Standard deviation of motif lengths of type $t_s t_d$:

$$\sigma^2_{t_s t_d} = \frac{\sum_{i=1}^{N_{t_s t_d}} L^{i^2}_{t_s t_d}}{N_{t_s t_d}} - \mu^2_{t_s t_d}$$

- Global diversity of the motifs (the graph regularity is the inverse of the diversity):

$$\text{Div} = \sum_{i \in t_s, j \in t_d} \sigma_{ij} N_i N_j$$

$$\text{Reg} = \frac{1}{\text{Div}}$$

where N_i and N_j are the numbers of operators of type i and j.

These weights (N_i and N_j) priviledge the minimum diversity of data transfers between the more frequently used operations in the graph.

The computation for the examples in Figure 8 yields:

- For the regular graph:
  - For the motifs of type $++$: $\{L^1_{++}, L^2_{++}, L^3_{++}\} = \{1,1,1\} \rightarrow \mu_{++} = 1, \sigma_{++} = 0$
  - For the motifs of type $+\times$: $\{L^4_{+\times}, L^5_{+\times}, L^6_{+\times}\} = \{1,1,1\} \rightarrow \mu_{+\times} = 1, \sigma_{+\times} = 0$
  - For the motifs of type $\times+$: $\{L^7_{\times+}, L^8_{\times+}, L^9_{\times+}\} = \{1,1,1\} \rightarrow \mu_{\times+} = 1, \sigma_{\times+} = 0$
  - For the motifs of type $\times\times$: $\{L^{10}_{\times\times}, L^{11}_{\times\times}, L^{12}_{\times\times}\} = \{1,1,1\} \rightarrow \mu_{\times\times} = 1, \sigma_{\times\times} = 0$
  - Graph regularity:

$$\text{Div} = 0 \rightarrow \text{Reg} = \infty$$

- For the irreuglar graph:
  - For the length set of motifs of type $++$: $\{L^1_{++}, L^2_{++}, L^3_{++}\} = \{2, 2, 1\} \rightarrow \mu_{++} = 1.66, \sigma_{++} = 0.24$
  - For the motifs of type $+\times$: $\{L^4_{+\times}, L^5_{+\times}, L^6_{+\times}\} = \{2, 1, 1\} \rightarrow \mu_{+\times} = 1.33, \sigma_{+\times} = 0.23$
  - For the motifs of type $\times+$: $\{L^7_{\times+}, L^8_{\times+}, L^9_{\times+}\} = \{1, 2, 2\} \rightarrow \mu_{\times+} = 1.66, \sigma_{\times+} = 0.24$
  - For the motifs of type $\times\times$: $\{L^{10}_{\times\times}, L^{11}_{\times\times}, L^{12}_{\times\times}\} = \{1,1,2\} \rightarrow \mu_{\times\times} = 1.33, \sigma_{\times\times} = 0.23$
  - Graph regularity:

$$\text{Div} = 15.04 \rightarrow \text{Reg} = 0.066$$

One can note another difference between these graphs, apart regularity: the number of machine cycles is quite different. Nevertheless, other examples show that with a constant number of machine cycles, differences between regularities can be significant.

4.4 Binding Algorithm

The second algorithm – still based on a Simulated Annealing process – performs simultaneously operator binding (allocation of functional unit instances to operations in the graph) and transfer resource minimization (registers and multiplexors). In [20] the binding algorithm has been extended in order to handle the geometric placement of modules (floorpan). In addition, this first version took into account the interconnect length between hardware modules within the cost function.

However, our main goal was to implement in the ALPHA synthesis system all our high level expertise in Image Processing algorithms optimization and to leave back to the low level CAD tools the technology dependent optimizations. Moreover, considering the different existing methods for placing and routing low-level cells (standard cells style, gate-array, parameterised cells) and the technology-independance constraint for the system, this phase has been shown very dependent on the particular CAD tool and not really efficient. The system described herein no longer offers this aspect of the synthesis.

For this optimization, transformations are of three types:

- **Move-Assign**: a new module instance is selected to implementing a randomly selected node. This instance is picked up in the candidate list of the node.
- **Move-Swap**: two instances of the same module implementing two nodes are swapped.
- **Move-Commut**: inputs of a node (if there is a couple) are swapped, if the concerned operation is commutative.

The minimized energy (the cost function) is expressed in terms of:

- The total number of registers.
- The total number of multiplexors.

At this step of the synthesis, the problem is represented as two internal structures. The first one sums up the interconnections between hardware modules (the interconnection matrix). All modules are listed in this structure (functional units and registers). The total number of multiplexor inputs (and therefore the number of multiplexors) can be extracted fom this matrix whenever it is needed (Fig. 9).

The second structure is a set of *slices*. Each slice is a set of functional nodes having resource conflicts in common (multiple non-exclusive similar operations executed on the same machine cycle, overlapped multicycled operations). By regrouping these nodes in the same structure, one can easily detect any module parallelism and therefore find a new instance candidate for a particular node. A particular slice contains not only a set of exclusive nodes (in terms of hardware sharing) but also a list of the available hardware instances for their implementation (an example of the slices is shown Fig. 10).

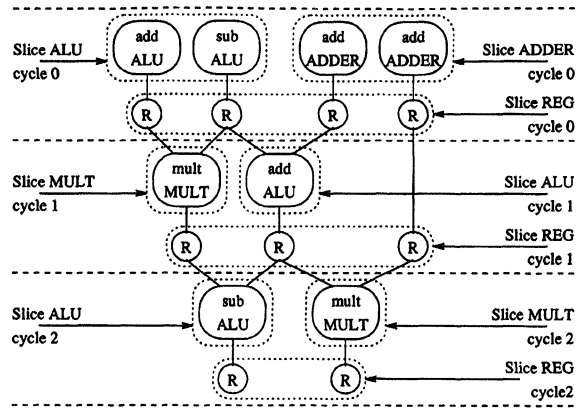

FIGURE 10 Example of a few slices in a control/data-flow graph.

4.5 Netlist Generation

The last phase of the synthesis aims at generating the low level (RTL) description of the architecture. A structural VHDL interface has been developed and feeds a layout generator tool (currently we use the COMPASS VLSI design framework). In order to make use of all the logic and gate minimization capabilities of such an environment, all the hardware modules are described by VHDL instances. A VHDL library (package) contains the descriptions corresponding to all the module databse entries. The controller specification is also generated and takes the form of a high-level state-based description of the FSM.

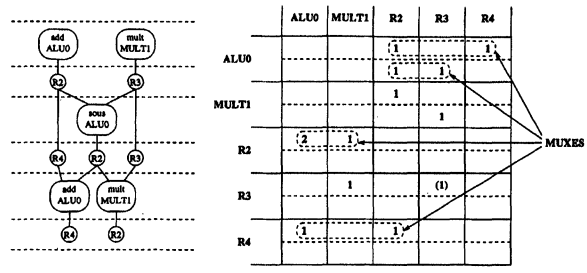

FIGURE 9 Multiplexors computation from the interconnection matrix. Brackets indicate that register R3 keeps its own value and does not need any input multiplexor.

Due to the application domain – embedded data-flow Image Processing systems – , the interface between the synthesized chip and its environment is done by using synchronous dual port FIFOs. These communication devices are controlled by the in-chip controller and regulate the chip throughput. Whenever an input FIFO is empty (or an output one is full) a wait state is generated until new data are available (a previous result is read by an external device).

We discussed in 4.4 that we did not integrate in our system the low level implementation dependent optimizations (floorplan area and interconnect length estimations). Of course, it is well known that an optimal representation of an architecture can not be reached automatically without taking into account such critical knowledge. But it appeared important for us to handle first high level considerations like behavioral complexity (in terms of number of modules, sharing resources, pipelining functional units, minimizing control complexity) rather than lower level interconnect or gate information.

5. EXPERIMENTAL RESULTS

The example shown here is a default detector on a model of human preattentive vision for pattern inspection.

In this algorithm, presented in [21], a default is defined as a local perturbation respective to a set of preponderant lines in terms of their edge direction. Figure 11 shows an example of misalignment detection.

Two data types are processed: the binary image of edge pixels and the image of their directions. A key parameter of the procedure is the edge orientation code accuracy (for instance 4 directions are processed here).

The algorithm runs as follows (Fig. 11): For each direction D_k and each edge pixel three tasks are performed:

1. Recording pixel directions, through a gaussian distribution centered in D_k, by their "distance" to direction D_k, followed by a spatial smoothing for noise reduction. Whence a local mean relative direction $\overline{Ld_k}$
2. Computing a global (for all edge pixels) value of these local relative directions: $\overline{Gd_k}$
3. The latter is used as a normalization for the local contribution. Let be:

$$C_c = \frac{\overline{Ld_k}}{\overline{Gd_k}} \tag{13}$$

Eventually, C_c represents a misalignment with respect to each direction D_k, smoothed and normalized.

Then, for all directions D_k, the maximum misalignment is taken and spatially smoothed again for stability. Here the convolution kernel size is parameterized considering domainant texture pattern features or desired default sizes. The ultimate detection is done by thresholding this maximum.

Figure 12 shows the edge detected input image, the image of misaligned pixels an the final binarized output image produced by the algorithm emulation on the Functional computer.

The interesting feature in this selected example is the hierarchical specification. It is illustrated by the separate description of the main process and the 1-direction macro (Fig. 13). The synthesis of the algorithm has been done by compiling separately the 1-direction macro and by including its hardware structure into the module database. In doing so, semantic level of available modules is shown to be extended by the synthesis system

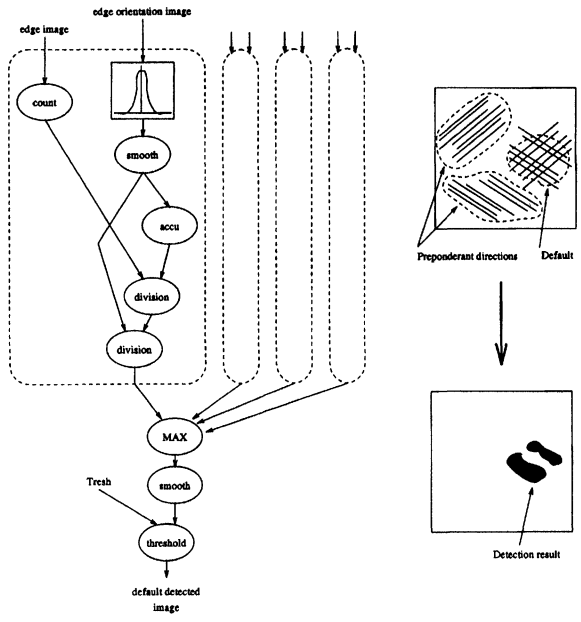

FIGURE 11 Detection of a local perturbation of predominant directions. The data-flow graph represented is the one appearing Figure 4 as direction macro.

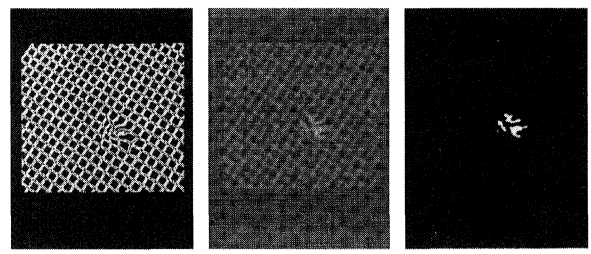

FIGURE 12 Image produced by the algorithm emulation.

```
// FP source code of main process

MAIN [
INPUT Image:PIXEL;
INPUT Thresh:PIXEL;
OUTPUT Res:PIXEL;
]
def dx = sous . [Image, Pixel_delay . Image];
def dy = sous . [Image, Trame_delay . Image];

def angle = xor . [sgn . dx, sgn . dy];
def adx = abs . dx;
def ady = abs . dy;
def m = max . [adx, ady];

def tg1 = atg_lut . [adx, ady];
def tg = norm . [tg1, angle];

def edge = thr . [Thresh, m];
def direction = select . [tg, edge];

def d0 = dir_macro(dir = 0) . [direction, edge];
def d1 = dir_macro(dir = 1) . [direction, edge];
def d2 = dir_macro(dir = 2) . [direction, edge];
def d3 = dir_macro(dir = 3) . [direction, edge];

def Res = max ! [d0, d1, d2, d3];
END
```

```
// FP source code of 1-direction macro
MAIN [
INPUT dir:PIXEL;
INPUT edge:PIXEL;
OUTPUT out:PIXEL;
]
def ex=gauss(param, 4).dir;

def eT = Trame_delay . ek;
def eTT = Trame_delay . eT;
def sT = add \ [ex, eT, eTT];
def eP = Pixel_delay . sT;
def ePP = Pixel_delay . eP;
def ed = add \ [sT, eP, ePP];
def a1 = select . [ed, edge];
def c1 = count16(outwidth = 8) . edge;
def c2 = accu(outwidth = 16) . a1;
def a2 = div(outwidth = 8) . [c2, c1];

def out = div . [a1, a2];
END
```

FIGURE 13 Hierarchical FP description of the default detector.

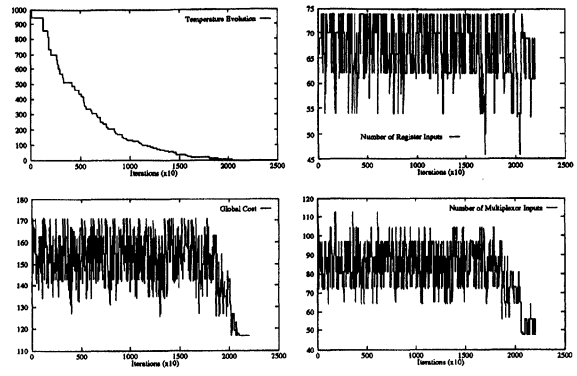

FIGURE 14 Evolution of the optimization parameters during the binding process for the 1-direction macro. One can denote that at the end of the process, multiplexor and register measures are orthogonal. While these parameters are still evolving, a cost minimum is found.

itself. Table I gives the final features of the two synthesis results.

First, the 1-direction macro has been synthesized as a structural netlist of register transfer level operators. Figure 14 shows the evolution of the optimization parameters during the synthesis step. The resulting netlist has been compiled, placed, routed and saved in the form of a macro-cell. Second, the main algorithm has been synthesized and each occurence of the 1-direction operation has been chosen to be implemented by the multicycled macro-cell. The clock frequency is adapted to the cell working frequency. Figure 15 gives the layout of the synthesized chip.

6. CONCLUSION

A high level synthesis system oriented towards Image Processing applications has been presented. First originality, within this environment, is the validation phase of the algorithm is tightly coupled with its integration phase. It allows the production of behaviorally correct systems and then considerably shortens their design cycle. Second, we use the data-flow graph regularity as a measure of the complexity of the control aspect of an architecture. Both features together turn out to be as efficient as expected after a few designs have been completed including medium complex chips like edge or region detectors. The ALPHA system is being used

TABLE I Results of the two synthesis steps for the 1-direction macro and the whole circuit

	1-direction macro	default detector
emulated on	59 DF-processors	294 DF-processors
technology	1 μm	1 μm
area	12 mm^2	63 mm^2 (core) 82 mm^2 (chip)
control steps	10	23
data frequency	3 Mhz	1.3 Mhz
clock freq.	30 Mhz	30 Mhz
control/datapath surface ratio	4%	15%

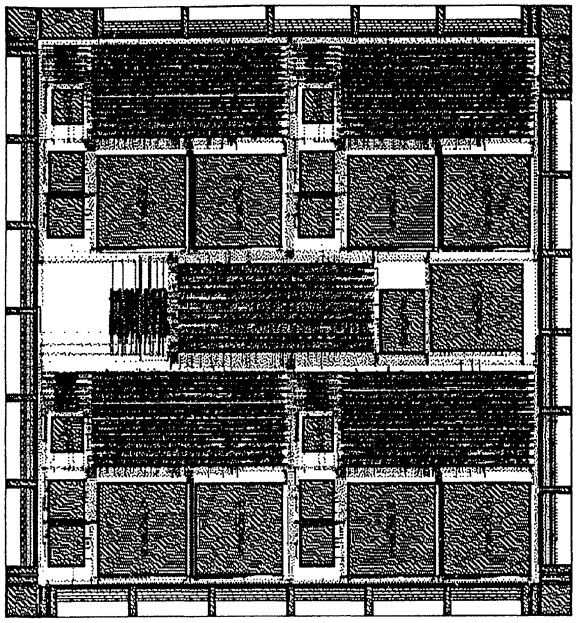

FIGURE 15 Layout of the synthesized default detector chip.

in a systematic manner for more complex architectures and compared to another design method experimented in our laboratory based on the mere derivation from the emulator architecture.

References

[1] Zavidovique, B., Serfaty, V. and Fortunel, C. Mechanism to capture and communicate image-processing expertise. *IEEE Software*, pp. 37–50, November 1991.

[2] Zavidovique, B., Fortunel, C., Quénot, G., Safir, A., Sérot, J. and Verdier, F. (1994). Automatic synthesis of vision automata. In Magdi A. Bayoumi, editor, *VLSI Design Methodologies for Digital Signal Processing Architectures*, pp. 261–318. Kluwer Academic Publisher.

[3] Sérot, J., Quénot, G. M. and Zavidovique, B. (1993). Functional programming on a data-flow architecture: applications in real-time image processing. In *International Journal of Machine Vision and Applications*, **7**, 44–56.

[4] Quénot, G. M., Kralji, I. C., Sérot, J. and Zavidovique, B. A reconfigurable compute engine for realtime vision automata prototyping, In *IEEE Workshop on FPGAs for Custom Computing Machines Napa, CA, USA*, pp. 91–100, April 1994.

[5] Backus, J. (1978). Can programming be liberated from the von-neumann style ? a functional style and its algebra of programs, *Communications of the ACM*, **21**.

[6] McFarland, Michael C., Parker, A. C. and Camposano, Raul. (1988). Tutrorial on high-level synthesis. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pp. 330–336.

[7] Knapp, David W. and Parker, Alice C. A unified representation for design information. In *Proceedings of the 7th International Symposium on Computer Hardware Description Languages and their Applications*, pp. 337–353, August 1985.

[8] Park, Nohbyung and Parker, Alice (1986). SEHWA: A program synthesis of pipelines. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pp. 454–460.

[9] Walker, Robert A. and Thomas, Donald E. Design representation and transformation in the System Architect's Workbench. In *Proceedings of ICCAD-87, Santa Clara, CA, USA*, pp. 166–169, (November 1987).

[10] Rim, Minjoong and Jain, Rajiv. (1992). Representing conditional branches for high-level synthesis applications. In *Proceedings of the 29th ACM/IEEE Design Automation Conference, Anaheim, CA, USA*, pp. 106–111.

[11] van Laarhoven, P. J. M. and Arts, E. H. L. (1988). *Simulated Annealing: Theory and Applications*. D. Reidel Publishing Company.

[12] Safir, A. and Zavidovique, B. Towards a global solution to high-level synthesis problems. In *Proceedings of European Design Automation Conference. Glasgow, Scotland, U. K.*, pp. 283–288, (March 1990).

[13] Safir, A. and Zavidovique, B. A solution to the high level synthesis problems. *The International Journal of Computer Aided VLSI Design*, **3**, 43–69, (January 1991).

[14] Potkonjak, Miodrag. and Rabaey, Jan. (1994). Optimization resource utilization using transformations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **CAD-13**, 277–292.

[15] Devadas, S. and Newton, A. R. (1989). Decomposition and factorization of sequential finite state machines. *IEEE Transactions on Compute-Aided Design of Integrated Circuits and Systems*, **8**(11).

[16] Sreenivasa Rao, D. and Kurdahi, Fadi J. (1992). Partitioning by regularity extraction. In *Proceedings of the 29th ACM/IEEE Design Automation Conference, Anaheim, CA, USA*, pp. 235–238.

[17] Sreenivasa Rao, D. and Kurdahi, F. J. Controller and datapath trade-offs in hierarchical RT-Level synthesis. In *Proceedings of the 7th International Symposium on High-Level Synthesis, Niagara-on-the-lake, Ontario, CA*, pp. 152–157, (May 1994).

[18] Verdier, F., Safir, A. and Zavidovique, B. A high level synthesis algorithm including control constraints. In *Microprocessing and Microprogramming, Proceedings EUROMICRO 92, Paris*, pp. 271–278, (September 1992).

[19] Paulin, Pierre G. (1989). Algorithms for high-level synthesis with area and interconnect constraints. In *Proceedings of the EURO ASIC'89, grenoble france*, pp. 144–158.

[20] Safir, A. and Zavidovique, B. Towards a global solution to high-level synthesis problems II. In *Proceedings of IFIP Working conference on Logic and Architecture synthesis, Paris, France*, pp. 151–158, (May/June 1990).

[21] Brecher, Virginia. (1992). New techniques for patterned wafer inspection based on a model of human preatentive vision. *SPIE Journal on Applications of Artificial Intelligence, Machine Vision and Robotics*, **1708**, 452–459.

Authors' Biographies

François S. Verdier has received a Ph.D. in Computer Science from the Electrical Engineering Department of PARIS XI University in 1995 and is working at the ETCA Perception System Laboratory in the field of High Level Synthesis of embedded architectures for Image Processing. His research interests are focused on high level description of algorithms and architectures, High Level Synthesis, formal verification of automatic derived architectures and optimizations techniques for integrated circuits. He is a student member of the ACM.

Bertrand Zavidovique is currently professor in the Electrical Engineering Department at PARIS XI University, and a scientific advisor at the DRET/ETCA for the problems of real-time processing and robotics, heading the Perception System Laboratory. He was invited at UC Berkeley-EECS, in 1986. His research interests include perception systems, the impact of their internal organization on external efficiency, real-time implementation, architecture and programming methods within the frame of circuit integration. He received a M.S. in mathematics form PARIS VII University in 1971, a Ph.D., in Computer Science from the University of Tours (France) in 1975, and a Doctorate of Sciences in Robot Vision from the University of Franche Comté (France) in 1981. Since then, he has published more than two hundred scientific papers in Image Processing, Sensor Fusion, Computer Architecture, Intelligent control and Learning.