An automated design flow for optimized implementation of real-time image processing applications on FPGA

Linda Kaouane, Mohamed AKIL, and Yves SOREL
Groupe ESIEE–Laboratoire A2SI,
BP 99 - 93162 Noisy-le-Grand, France
{kaouanel, akilm}@esiee.fr
INRIA Rocquencourt–OSTRE,
BP 105 - 78153 Le Chesnay Cedex, France
yves.sorel@inria.fr

Abstract—**As the size and complexity of high performance signal, image and control processing algorithms is increasing continously, the implementations cost is becoming an important factor. This paper addresses this issue and presents an efficient rapid prototyping methodology to implement such high performance algorithms using reconfigurable hardware. Such reconfigurable architectures, like FPGAs, provide all the benefits of hardware acceleration while retaining the flexibility of programmation. The proposed design methodology follows a seamless design flow of graph transformations from the specification to the final implementation, which is supported by SynDEx[1], a system level CAD software tool.**

I. Introduction

THE real-time implementation of signal, image and control processing algorithms in embedded system is a difficult task, and generate generally complex programs hard to test and debug. This explain the need for automated design flow and rapid prototyping methodologies to implement efficiently such complex algorithms on the target architectures.

To help cope with this need, the OSTRE team of INRIA has developped the AAA (Algorithm-Architecture Adequation) rapid prototyping methodology which helps the real-time application designer to obtain rapidly an efficient implementation of his application algorithm on his heterogenous multiprocessors architecture and to generate automatically the corresponding distributed executive [2]. This methodology is based on graphs models to modelize the algorithm, the distributed architecture as well as the implementation which is formalized in terms of transformations applied on the previously defined graphs.

Indeed the real-time and embedded constraints may be so strong that the available high performance processors are not sufficient. This leads to the use, in complement of processors, of specific components like ASIC or FPGA. The aim of our research is to extend the AAA methodology to the implementation of real-time image and signal processing applications on reconfigurable circuits. This extension uses a single factorized graph model, from the algorithm specification down to the architecture implementation, through optimizations expressed in terms of defactotization transformations [3].

This paper will briefly presents the transformation flow used in this extension to automate the hardware implementation process of an application algorithm on reconfigurable circuits. First, we present in section II the factorized data dependence graph model proposed to specify the application algorithm. Then we present, in section III, the implementation model especially the principles allowing to automate the synthesis of both data and control paths from the algorithm specification. The principles of optimization by defactorization are shown in section IV. Next a motivating example of edge detector used to illustrate the proposed methodology is described in section V. Finally, section VI concludes and discusses future work.

[1]Software tool that support AAA (http://www-rocq.inria.fr/syndex)

II. Factorized Data Dependence Graph model

According to the AAA methodology, the algorithm model is an extention of the directed acyclic graph of operations (DAG), where each node models an operation (more or less complex, e.g. an addition or a filter), and each oriented hyperedge models a data, produced as output of a node, and used as input of an other node or several other nodes (data diffusion).

In image and signal processing, this specification model based on data dependence between implementable operations frequently generates repetitions of operation patterns (identical operations that operate on different data). To reduce the size of the specification and to highlight these regular parts we use a graph factorization process which consists in replacing a repeated pattern, i.e. a subgraph, by only one instance of the pattern, and in marking each edge crossing the pattern frontier with a special "factorization" node, and the factorization frontier itself by dashed line crossing these nodes.

The type of the factorization node depends on the way the data are managed when crossing a factorization frontier. Then a factorization node may be:

- a Fork node (F): array partition in as many subarrays as repetitions of the pattern;
- a Join node (J): array composition from results of each repetition of the pattern;
- a Diffusion node (D): diffusion of a data to all repetitions of the pattern;
- an Iterate node (I): data dependence between iterations of the pattern.

This extention provides specification of loops through factor-

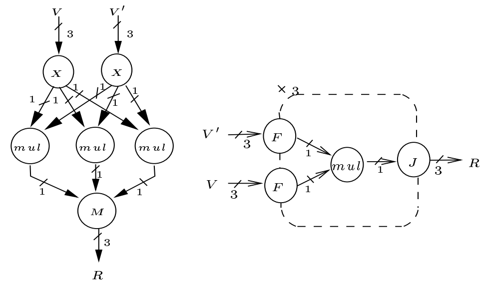

Fig. 1. Example of algorithm graph

ization nodes leading to an algorithm model, that we call Factorized Data Dependences Graph (FDDG) [3]. This algorithm graph may be specified directly by the user using the graphical or textual interface of the SynDEx software or it may be generated by the compiler from high level specification languages. Such synchronous languages, Esterel, Lustre, Signal perform formal verifications in terms of events ordering in order to reject specifications including deadlocks [1].

Note that from the algorithm specification point of view, the factorization reduces only the size of the specification, without any modification of its semantics. For instance, the graphs in Fig.1 specify both the same vector product of two integer vectors V and V' of dimension 3, the one in Fig.1.a is a non factorized data dependence graph and the one in Fig.1.b is the equivalent (from the specification point of view) factorized data dependence graph. In Fig.1.a the nodes X are an array-decomposition operation which separates its input array V (respec. V') into its elements, and the node M is an array-composition operation which re-assembles its input elements in a data output array R. Althought apparently, Fig.1.a and Fig.1.b are not the same graph (different nodes and edges), they have the same semantics: apply the product operation mul as many times (3) as there are elements in the vectors to multiply. However, from the implementation point of view, the factorization allows all the possible implementations, from the entirely parallel one to the entirely sequential one, with all the intermediate cases mixing both sequential and parallel. The factorized graph of Fig.1.b may be implemented of one of all its possible implementations. That is to say, an implementation where the three multiply operators will be executed sequentially through an iteration, or will be executed all in parallel, or two of them will be executed in parallel and executed sequentially with the third one, etc. Obviously, each of these implementation will have different characteristics in terms of area and response time.

III. Implementation model

Given an algorithm graph, we transform it to an implementation graph. This transformation is composed of the generation of the data-path graph responsible for the core of the computation and of the control-path graph to generate the appropriate control signals. Data-path transformations are quite simple, but control-path transformations are not trivial and require to build first a neighborhood graph.

A. *Neighborhood graph : Relationship between frontiers*

Every factorization frontier may be a consumer (located downstream) or/and a producer (located upstream) relatively to another frontier according to the data dependences. Two frontiers are neighbor if there is at least one relation of direct dependence that does not cross a third frontier. Based on these neighborhood relations, we build a graph where each node represent the factorization frontier and each oriented edge represent the data flow between factorization frontiers. The edge orientation describes the consumption/production relation: an edge starts at a producer and ends at a consumer.

In the case of a sequential implementation of factorization nodes, every factorization frontier, called FF, separates two regions, the first one called "fast", being repeated relatively to the second one, called "slow". These slow and fast sides of a frontier are due to the difference of frequency of the data transfer on each side of the factorization frontier. Every node of the neighborhood graph is then subdivided in four parts (see Fig.2):

- slow-downstream: "slow" side of a consumer FF;
- fast-upstream: "fast" sideof a producer FF;
- fast-downstream: "fast" side of a consumer FF;
- slow-upstream : "slow" side of a producer FF.

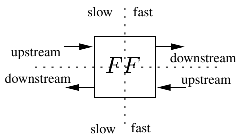

Fig. 2. Node of neighborhood graph representing a factorization frontier FF

Then the neighborhood graph, deduced automatically from the FDDG, is used during the implementation to establish the control relationships between frontiers.

B. *Circuits synthesis*

High-level circuit synthesis transforms high-level behavioral representation into a register-transfert-level structural description (RTL) containing both the data path and control path. It is known that the automation of this synthesis process reduces significantly the development cycle of the circuit since it allows the exploration of different hardware implementations, seeking for an ideal compromise between the area and the response time of the circuit. Afterwards, we will present the principles used to automate this step and to generate automatically the data and the control paths of the circuit, from the factorized data dependence graph and the neighborhood graph.

B.1 Data path synthesis

The hardware implementation of the factorized data dependence graph consists in providing a matching operator for every operation node, and a matching connection between operators for each data dependences edge between the corresponding operations. The resulted graph of operators and their interconnections compose the data path of the circuit. The matching operator is a logic function in the case of an operation node, or it is composed of a multiplexer/demultiplexer and/or registers in the case of a factorization node.

B.2 Control path synthesis

The control path corresponds to the logic functions that must be added to the data path, in order to control the multiplexers/demultiplexers and the transitions of the registers composing the factorization operators. It is then obtained by data transfer synchronization between registers. However, two conditions must be satisfied to allow a register to change state: the new upstream data to the register must be stable, and all downstream consumers of the register must have finished the utilization of previous data. If moreover upstream data of a circuit comes from various producers with different propagation time, it is necessary to have a synchronized circuit. This synchronization is possible through the use of a request/acknowledge communication protocol.

Given that these operators are gathered in factorization frontier and their data consumption and production are done in a synchronous way at the level of the frontier, the generated control must be a local control at each frontier. We propose then a control system where each factorization frontier will have its own control unit (Fig.3). This delocalized control approach allows the CAD tools used for the synthesis of the generated VHDL design to place the control units closer to the operators to control rather then a centralized control approach.

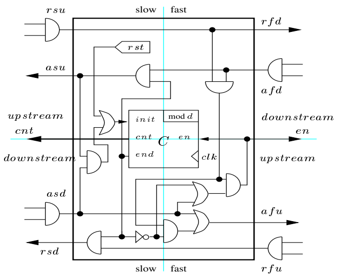

Fig. 3. Control unit

This control unit contains a counter C with d states (corresponding to the d factorized repetitions) and an additional logic function in order to generate, in the one hand the communication protocol between frontiers (the slow and fast, request and acknowledge signals at the upstream and downstream sides), and in the other hand the counter value (cnt) and the enable signal (en), that control the frontier operators: $F,\ J\ and\ I$.

IV. Implementation optimization: principles

For a given algorithm graph, there is a large, but finite, number of possible implementations which are more or less defactorized, and among which we need to select the most efficient one, i.e which satisfies real-time constraints (upper bound on latency) and which uses as less as possible the hardware ressources, number of Configurable Logic Blocks for FPGA. This optimization problem is known to be NP-hard, and its size is usually huge for realistic applications. This is why we use heuristics guided by their cost function in order to compare the performances of the different defactorizations of the specification. These heuristics, using tricks related to practice, allows us to explore only a small subset of all the possible defactorizations into the implementation space. These cost functions take into account the characteristics of an implementation in term of hardware ressources required (number of CLBs) and latency.

V. Implementation of deriche edge detector

As we are interested in image and signal processing applications, we will use the Garcia-Lorca optimal implementation of Deriche edge detector to validate our implementation design flow. This motivating example stands as a fundamental step in image segmentation process and illustrates perfectly many problems related to the real-time implementation.

A. *filter presentation*

The Garcia-Lorca optimal implementation of Deriche edge detector filter (Fig.4) comprises two mains steps:

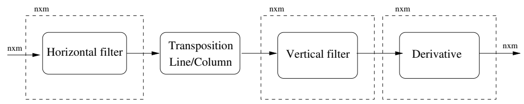

Fig. 4. Algorithm of deriche filter

- A vertical and horizontal smoother: for the horizontal smoothing of a line, we apply first two passes of the first order causal smoothing descibed by the following equation :

$$y(n) = (1-\gamma)x(n) + \gamma y(n-1)$$

($x(n$ is the n^{th} pixel of the current line, $y(n)$ the n^{th} filtred pixel). Then, we apply two passes of the first order anticausal smoothing descibed by the following equation:

$$y(n) = (1-\gamma)x(n+1) + \gamma y(n+1)$$

where γ is a constant between 0 and 1.
The whole horizontal smoothing filter (two passes for causal smoothing filter and two passes for anticausal smoothing filter) can be seen as 4 passes of the same causal filter if we invert the line elements (Fig.5). We proceed, in the same way, with the vertical filter;

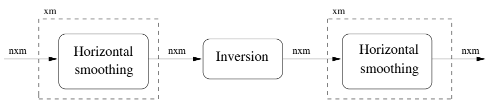

Fig. 5. Horizontal filter

- Derivative filter: is computed by the convolution of the following 2x2 kernels, R_h and R_v :

$$R_h \begin{pmatrix} -1 & 1 \\ -1 & 1 \end{pmatrix} \quad et \quad R_v \begin{pmatrix} -1 & -1 \\ +1 & +1 \end{pmatrix}$$

B. *Implementation flow*

The proposed seamless transformation flow used for the implementation is briefly summerized in Fig.6. In fig.7 we present the hierarchical top-down graphical specification of the Garcia-Lorca filter through the software tool SynDEx: The top window describes the algorithm specification given in Fig.4, the next window the horizontal filter given in Fig.5 and the following two windows specify both the first order causal smoothing filter, the last window specifies the derivative filter.

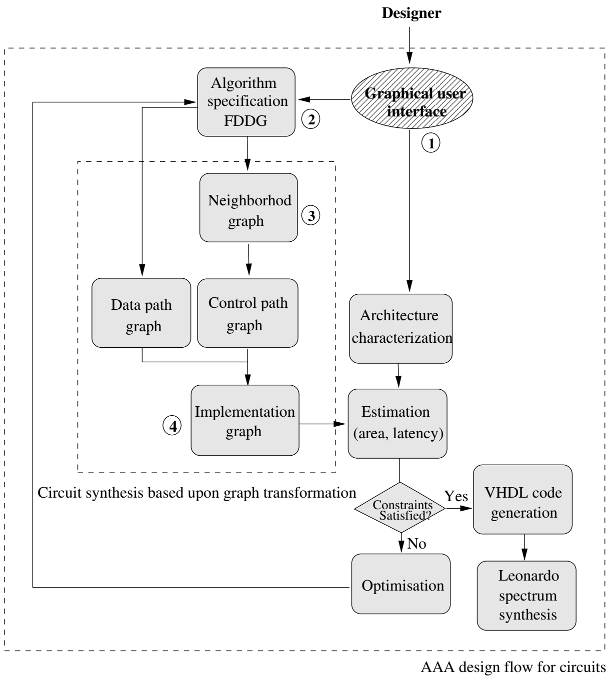

Fig. 6. Design flow methodology

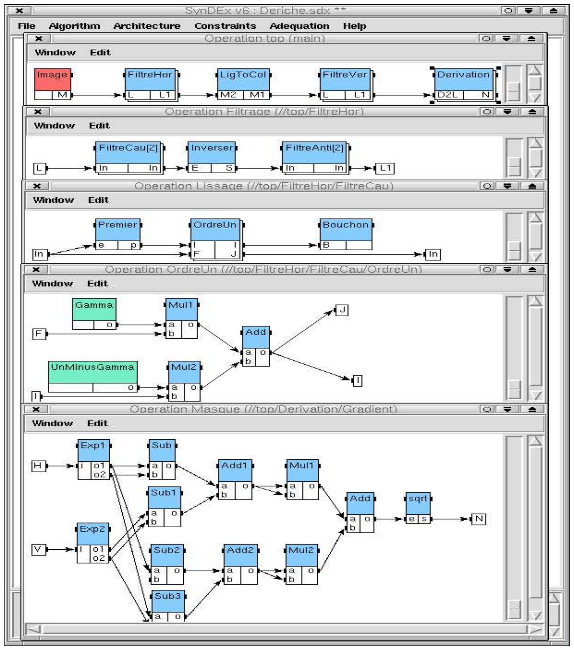

Fig. 7. Screen snapshot of SynDEx specifying the filter : step1

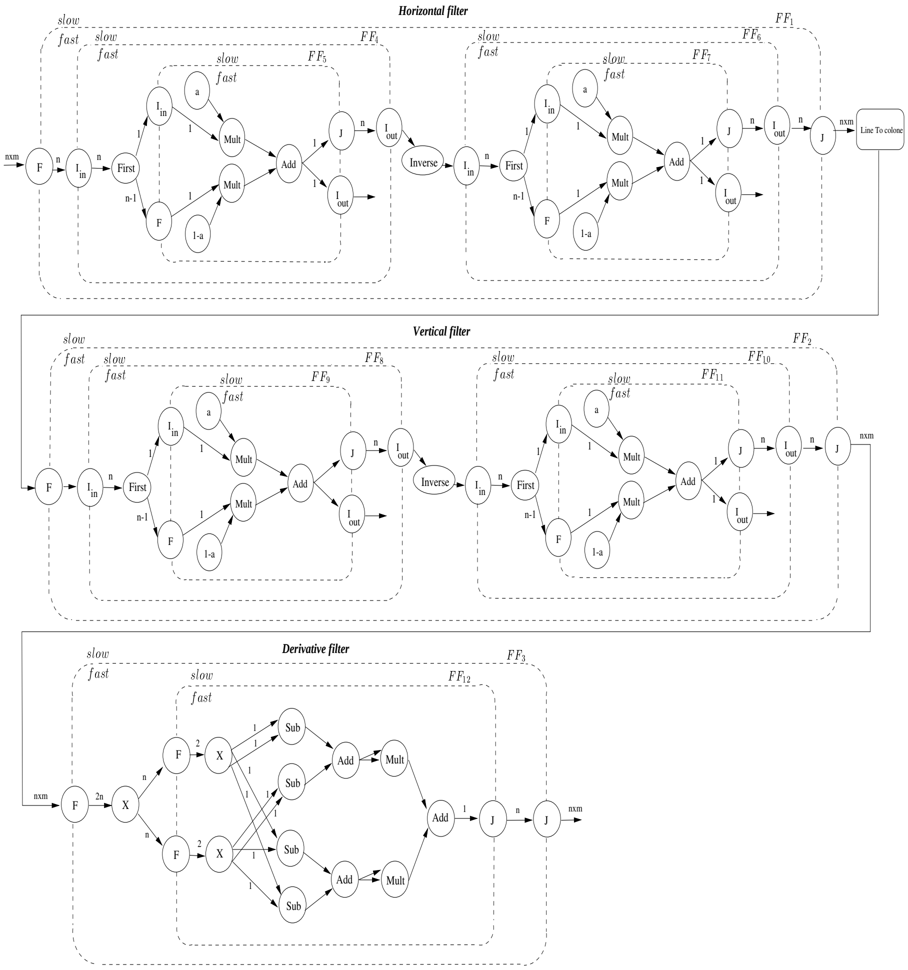

Fig. 8. FDDG of the deriche filter: step 2

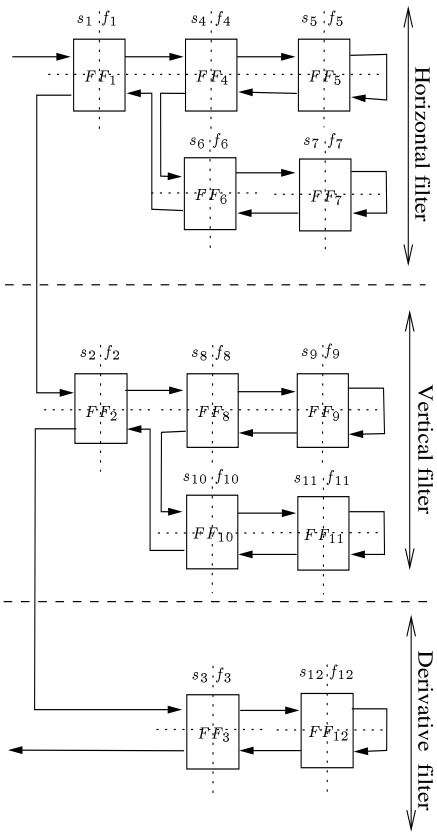

Fig. 9. Neighborhod graph: step 3

The corresponding factorized data dependence graph depicted in (Fig.7) is then used to deduce the neighborhood graph between the factorization frontiers $FF_i for (i = 1..12)$ as shown in (Fig.9).

Finally the generated implementation graph containing both data and control paths (Fig.7) is then characterized in order to estimate time and area performance. This estimation is based upon the characterization of the basic operators mult, add, sub, Fork, join,...implemented on FPGA *Virtex 300 BG 432*.

Tab.11 show results obtained by our optimization heuristic with various times constraints. For example, with a constaint latency of 1400*ns*, the heuristic use a defactorized implementation of the derivator frontier FF_{12} by a factor equal to 3. These results presented in function of, the area (hardware ressources: number of CLBs) and the latency in *ns* demonstrate that the most defactorized solutions allow to reduce the latency of the implementation, but they increase the number of required hardware ressources.

VI. Conclusions

We have presented a flow of transformations that lead seamlessly to the generation of a complete VHDL design corresponding to the implementation of an application specified by Factorized Data Dependence Graph model. We validated the proposed design flow on several examples representative of low-level image processing such as mean filtering[3], edge detection operators : Deriche, Sobel,...

This work is part of the extension of the AAA methodology implemented in the software SynDEx to support implementation on reconfigurable circuits. It has allowed us to carry out an automatic generator of structural synthesizable VHDL for mono-FPGA (one FPGA) architectures, that has been added to SynDEx [4]. The generated VHDL code which corresponds to the optimized FPGA implementation obtained by successive defactorizations of the factorized algorithm graph, is then used by a CAD tool in order to generate the netlist needed for the FPGA configuration.

Currently, we are extending this automated hardware implementation process to multi-FPGAs architectures. To support such architectures, the optimization heuristic will adress both defactorization and partitioning

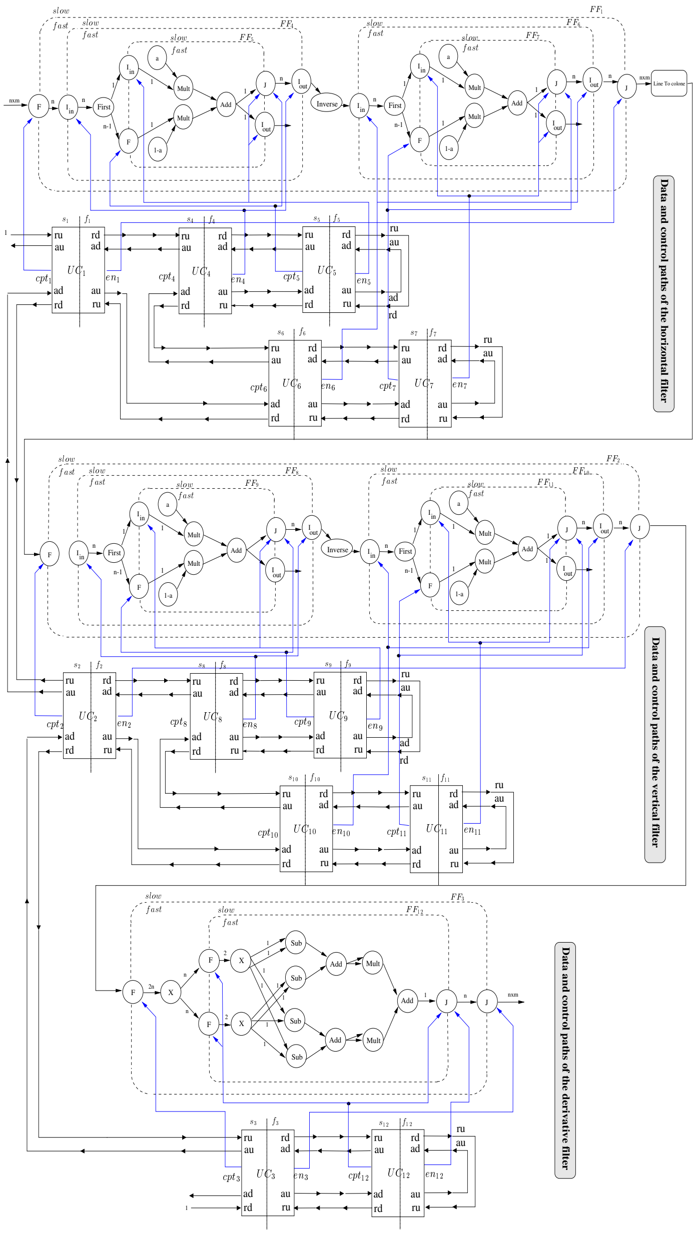

Fig. 10. Implementation graph: step 4

Implementation	*Constraint* (ns)	*Area* (CLB)	*Latency* (ns)
Fully.Factorized spec	2000	1406	1961
Part.Defact.FF_{12} by 2	1800	1346	1453
Part.Defact.FF_{12} by 3	1400	1444	1235
Part.Defact.FF_{12} by 4	1200	1384	1162
Fully.Defact.FF_{12}	1000	2344	944

Fig. 11. Implementation results on FPGA

References

[1] N. Halbwachs. *Synchronous programing of reactive systems*. Kluwer Academic Publishers, Dordrecht Boston, 1993.

[2] T. Grandpierre, Y. Sorel, *From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations*. First ACM & IEEE Intl. Conference on formal methods and models for codesign. MEMOCODE'03, june 24th-26th, 2003, mont saint-michel, france.

[3] A. F. Dias, C. Lavarenne, M. Akil, Y. Sorel. *Optimized implementation of real-time image processing algorithms on field programmable gate arrays*. Proc. of the 4th Intl. Conference on Signal Processing, Beijing, Oct. 1998.

[4] R. Vodisek, M. Akil, S.Gailhard, A.Zemva *Automatic Generation of VHDL code for SynDEx v6 software*. Electro technical and Computer Science conference, Portoroz, Slovenia, september 2001.