From algorithm graph specification to automatic synthesis of FPGA circuit: a seamless flow of graph transformations

Linda Kaouane[1], Mohamed Akil[1], Yves Sorel[2], and Thierry Grandpierre[1]

[1] Groupe ESIEE–Laboratoire A2SI, BP 99 - 93162 Noisy-le-Grand, France
{kaouanel, akilm, grandpit}@esiee.fr
[2] INRIA Rocquencourt–OSTRE, BP 105 - 78153 Le Chesnay Cedex, France
yves.sorel@inria.fr

Abstract. The control, signal and image processing applications are complex in terms of algorithms, hardware architectures and real-time/embedded constraints. System level CAD softwares are then useful to help the designer for prototyping and optimizing these applications. These tools are oftently based on design flow methodologies. This paper presents a seamless design flow which transforms a data dependence graph specifying the application into an implementation graph containing both data and control paths. The proposed approach follows a set of rules based on the RTL model and on mechanisms of synchronized data transfers in order to transform automatically the initial algorithmic graph into the implementation graph. This transformation flow is part of the extension of our AAA (Algorithm-Architecture Adequation) rapid prototyping methodology to support the optimized implementation of real-time applications on reconfigurable circuits. It has been implemented in SynDEx[3], a system level CAD software tool that supports AAA.

1 Introduction

The increasing complexity of signal, image and control processing algorithms in embedded applications requires high computational power to meet real-time constraints. This power can be achieved by high performance mixed hardware architectures built from different types of programmed components (RISC or CISC processors, DSP,..) to perform high level tasks and/or specific components (dedicated boards, ASIC, FPGA,...) used to perform efficiently low level tasks such as signal and image processing and devices control. Implementing these complex algorithms on such distributed and heterogenous architectures while verifying the severe real-time constraints is generally a difficult and complex task. This explain the need for dedicated high level design environnement based on efficient system-level design methodology to help the real-time application designer to solve the specification, validation and synthesis problems [1].

[3] http://www-rocq.inria.fr/syndex

In order to cope with these increasing needs, we have developped the AAA rapid prototyping methodology [5] wich helps the real-time application designer to obtain rapidly an efficient implementation of his application algorithm on his heterogenous multiprocessor architecture and to generate automatically the corresponding distributed executive. This methodology is based on an unified model of factorized graphs [2], as well to modelize the applicative algorithm and the multicomponent architecture, than to deduce the possible implementations in terms of graphs transformations.

Based on this model, we have extended AAA methodology to support the implementation of real-time 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 defactorization transformations applied to the algorithmic graph. This optimization aims to satisfy the real-time constraints while minimizing the required hardware resources [4]. In prospect, this extension is expected to allow the AAA methodology to be used for optimized hardware/software codesign.

In this paper, we focus on the rules used to synthesize both the data and the control paths of the circuit corresponding to an algorithm specified as a factorized data dependence graph. It is known that control path synthesis is more difficult to carry out than data path synthesis. We show here that it is possible to synthesize the control path in a secure and systematic way by using a technique of data transfers synchronization based on the RTL model. This approach allows us to carry out an automatic generator of synthesizable VHDL in a simple way. The remainder of the paper is organized as follows: in the next section, we briefly present the transformation flow used by our extented methodology to automate the hardware implementation process of an application algorithm on reconfigurable circuits. In section 3, we present the factorized data dependence graph model proposed to specify the application algorithm. As critical portions of control, signal and image processing algorithms often consist of regular computations generally expressed as nesteed loop, we will use a motivating example of matrix-vector product to illustrate the proposed transformation design flow. Section 4 gives rules to automate the synthesis of data and control paths extracted from the algorithm specification while the principles of optimization by defactorization are shown in section 5. We also show in section 6, the results of the implementation of the matrix-vector product algorithm onto a *Xilinx* FPGA following these rules. Finally, section 7 concludes and discusses future work.

2 AAA methodology for circuits

Given an algorithm graph specifying the application, we transform it to an implementation graph following a set of graphs transformations as described in Fig.1. This seamless transformation flow is composed of the generation of the data-path graph and the control-path graph. Data-path transformations are quite simple, but control-path transformations are not trivial and require to build first a neighborhod graph. Finally the implementation graph containing both data and control graphs is charaterized in order to estimate time and area performance. If the deduced implementation does not meet the user specified constraints, we apply a defactorization process in order to reduce the latency by increasing the hardware ressources. Since there is a large but finite number of possible defactorized implementations with different characteristics (FPGA area required, latency,..) among which we need to select the most efficient one, we need to use heuristics guided by their cost function. We then generate automatically the VHDL code corresponding to the resulting optimized implementation.

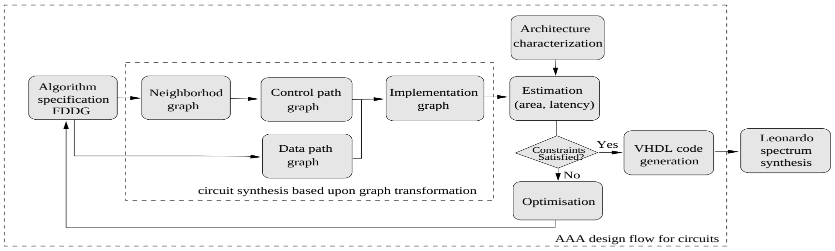
Architecture characterization
Algorithm specification FDDG
Neighborhod graph
Control path graph
Implementation graph
Estimation (area, latency)
Data path graph
Constraints Satisfied?
Yes
No
VHDL code generation
Leonardo spectrum synthesis
circuit synthesis based upon graph transformation
Optimisation
AAA design flow for circuits

Fig. 1. The AAA methodology for circuits

3 Algorithm model

The algorithm specification is the starting point of the process of hardware implementation of an algorithm application onto an architecture. According to the AAA methodology, the algorithm model is an extention of the directed data dependence graph (direct acyclic hypergraph 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). The extended model provides specification of loops through factorization nodes (fork, join, iterate, diffuse), leading to an algorithm model called Factorized Data Dependence Graph (FDDG) [4]. This algorithm graph may be specified directly by the user or it may be generated from high level specification languages such as the synchronous languages (Esterel, Signal,...), which perform formal verifications in terms of events ordering in order to prevent dead-locks [8].

3.1 Factorized Data Dependence Graphs Model

As described in [2], an algorithm specification contains regular parts (repetitive subgraph) and non-regular parts. This specification must be independent of all the constraints related to the hardware implementation, it then requires that the designer decomposes the algorithm into implementable operations. However, this decomposition frequently generates repetitions of operation patterns (identical operations that operate on different data). In fact, these spatial repetitions are usually reduced by a factorization process to reduce the size of the specification and to highlight its regular parts. Graph factorization 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 a dashed line crossing these nodes. The type of the factorization node depends on the way the data are managed when crossing a factorization frontier, it 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) or an Iterate node 'I' (data dependence between iterations of the pattern).

Note that from the algorithm specification point of view, the factorization reduces only the size of the specification, without any modification of its semantics. However, from the implementation point of view, the factorization describes also in intention all the possible implementations, from the entirely parallel one to the entirely sequential one, with all the intermediate cases mixing both sequential and parallel. Obviously, each of these implementation will have different characteristics in terms of area and response time.

3.2 Neighborhood graph

Every factorization frontier may be a consumer (located downstream) or/and a producer (located upstream) relatively to another frontier according to the data dependences relating them. Two frontiers are neighbors if there is at least one relation of direct dependence that does not cross a third frontier.

Based on these neighborhood relations between the factorization frontiers, we build a neighborhood graph of any algorithm graph. The nodes of such graph represent the factorization frontiers and the oriented edges 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" side of a producer FF;
- fast-downstream: "fast" side of a consumer FF;
- slow-upstream : "slow" side of a producer FF.

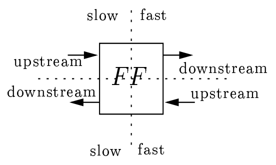

Fig. 2. Node of neighborhood graph for a frontier FF

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

3.3 Example: Specification of MVP (Matrix-Vector Product)

We now use a Matrix-Vector Product example (MVP) to illustrate the algorithm model of specification and its use for the building of the neighborhood graph. The choice of this example was motivated for a purely academic interest: first, it is so easy to understand and it presents regular computation on different array data which highlight the use of the factorization process. Second, this example concentrates its computation in nested loops that manipulate multidimensional array data structures and such complex but regular computations are of interest in signal and image processing applications. So the MVP of one matrix $A \in R^m \times R^n$ by a vector $B \in R^n$ gives a vector $C \in R^m$, and can be written in a factorized form as follows: $C = \left[\sum_{j=1}^{n} a_{ij} b_j \right]_{i=1}^{m}$

This equation allows us to obtain the graph corresponding to the algorithm specification of the factorized MVP (Fig. 3). The interface with the physical environment is delimited by input (F_A^∞ et F_B^∞) and by output (J_C^∞). It corresponds to the factorization frontier of the infinitely repeated pattern of the graph (FF_1) due to the reactive aspect of embedded real-time applications. The square brackets $[\ ]_{i=1}^{m}$ correspond to a second frontier (FF_2), delimited by factorization nodes of a finitely repeated pattern. This frontier selects the m lines of the matrix A (F_{21}), diffuses the vector B (D_{21}) and collects the result vector C (J_{21}). The functor $\sum_{j=1}^{n}$ corresponds to a third frontier (FF_3), also delimited by factorization nodes of a second finitely repeated pattern. This frontier selects the a_{ij} elements of the ith line of the matrix A (F_{31}) and the elements b_j of the vector B (F_{32}) and it supplies the result of the sum of products between a_{ij} et b_j for every line of matrix A (I_{31}). The "slow" and "fast" sides of each frontier are labeled "s" and "f", respectively.

The neighborhood graph between factorization frontiers, obtained from the factorized data dependence graph specifying the MVP algorithm, is shown in Fig.4. Because the factorization frontier FF_1 is infinite, it does not have neighbor on its "slow" side which corresponds to the physical environment. FF_1 is, at the same time, a producer (edges A and B) and a consumer (edge C) compared to FF_2. FF_2 is also a producer (edges A_i and B) and a consumer (edge C_i) compared to FF_3. FF_3 is a producer and a consumer, compared to itself through the arithmetic operations *mul* and *add*.

4 Circuit synthesis from a neighborhood graph

High-level circuit synthesis transforms a high-level behavioral specification into a register-transfert-level implementation (RTL) [6]. The resulting RTL design

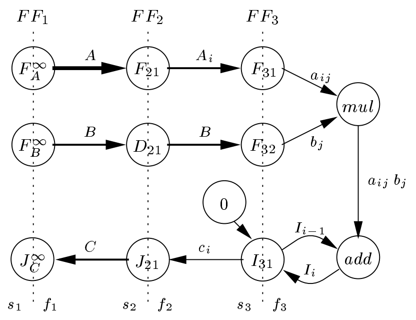

Fig. 3. Factorized data dependence graph of MVP

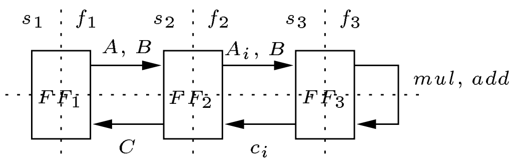

Fig. 4. Neighborhood graph of MVP: relations between frontiers

containing both the data path and control path can then be synthesized using logic synthesis tools that map components such as adders, multipiers,... to gates, perform optimization and finally generate the netlist of the final design. The automation of this synthesis process reduces significantly the development cycle of the circuit, and allows the exploration of different hardware implementations, seeking for an efficient compromise between area and response time of the circuit. Afterward we will present the rules used to automatically generate the data path and the control path of the circuit, from the factorized data dependence graph and the neighborhood graph.

4.1 Data path synthesis

The hardware implementation of operations consists in providing for every node of the factorized data dependence graph a matching operator (by instanciating the corresponding component of a VHDL library). The matching operator is a logic function in the case of an operation node, or it is composed of a multiplexer and/or registers in the case of a factorization node (F is implemented by a multiplexer, J by a demultiplexer with a memory array, I by a register with initialization value,..). The hardware implementation of the data dependences between operations consists in providing, for each edge of the FDDG graph, a matching connection between operators. The resulting graph of operators and their interconnections compose the data path of the circuit.

4.2 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 utilisation 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. Consequently, the synchronization of the circuit implementing the algorithm is reduced to the synchronization of the request/acknowledge signals of the set of factorization operators.

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. This delocalized control approach allows the CAD tools used for the synthesis to place the control units closer to the operators to control rather then a centralized control approach.

Control unit As mentioned above, each factorization frontier has upstream and downstream relations on both sides, “slow” and “fast”. The relations between upstream/downstream and request/acknowledge signals on both sides of a frontier are implemented by the “control unit” of the factorization frontier (Fig.5). 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. The counter value (cnt) controls the frontier operators: F, J and I. The enable signal (en) determines the clock cycles where the registers of the frontier operators (J, I, F^{∞}”sensor”, J^{∞}”actuator”) will change state. Note that, the signal ($init$) resets the counter while the signal (end) indicates that the counter is in its last state ($d-1$).

All the other signals are the request (r) and acknowledge (a) signals generated by the frontier(s) located upstream or diffused to the frontier(s) located downstream. They are separated in two groups: those which relate to the frontier(s) located on the ”slow” side and those which relate to the frontier(s) located on the ”fast” side, corresponding to the four parts of the control unit: slow-upstream (su), slow-downstream (sd), fast-upstream (fu) and fast-downstream (fd).

As mentioned above, the control path is mainly composed of the set of control units associated to the factorization frontiers of the application algorithm graph. These control units can then be inter-connected in an automatic way based on relationships between the factorization frontiers deduced from the neighborhood graph. In this control graph, the nodes correspond to the control units and the edges correspond to the request signals transmitted between the control units. The acknowledge signals are transmitted, in the opposite direction of the associated request signals, between the same control units. When several signals are at the same input of a control unit, one takes the conjunction by a logical gate AND.

5 Implementation optimization: principles

As previouly mentioned, the optimized implementation of a factorized algorithm graph onto an application specific integrated circuit or a FPGA, is formalized in terms of graph transformations, i.e defactorization. When we defactorize a graph we expect to reduce the latency by increasing the number of hardware ressources. Thus, the implementation space, which must be explored in order to find the best solution, is composed of all the possible defactorizations of a factorized algorithm graph. For instance, for a given algorithm graph with n frontiers, we have at least 2^n defactorized implementations. Moreover, each frontier can be partially defactorized: a factorization frontier of r repetitions can be decomposed in f factorization frontiers of r/f repetitions. Consequently, 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, logic gates for ASIC and number of Configurable Logic Blocks CLB 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 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: hardware ressources required (number of gates or CLBs) and latency.

6 Example: Synthesis of MVP implementation on FPGA's circuit

The Fig.6 represents the hardware implementation of the factorized MVP corresponding to the algorithm specification given in Fig.3. The data path is composed of the factorization frontier operators ($F_{i,j}$, $D_{i,j}$, $J_{i,j}$ and $I_{i,j}$) and of the combinatorial operators *mul* and *add* delimited by a dotted box. The control path is composed of the control units UC_1, UC_2 and UC_3, and of the control signals r (request), a (acknowldge), *cpt* and *en*. The interconnections between the request and acknowledge signals, is based on the relationships between the factorization frontiers, namely the neiborhood graph (Fig.4) built from the algorithm graph.

In Fig.7 we present the hardware implementation of a defactorized solution corresponding to the partial defactorization of the frontier FF_2 by a factor of 2. The $FF2$ frontier has been replaced by two frontiers FF_{2a}, FF_{2b} being repeated $m/2$ times (m : factor of repetitions of FF_2). The factorization frontier FF_3 remain inchanged but it has been duplicated (FF_{3a}, FF_{3b}) due to the partial defactorization of FF_2. The data path is then composed of the factorization frontier operators, the combinatorial operators delimited by dotted boxes and of the operators X_1 (array-decomposition operation), M_1 (array-composition operation). The control path is composed of the control units UC_1, UC_{2a}, UC_{2b},

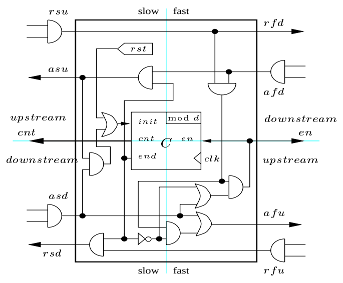

Fig. 5. Control unit

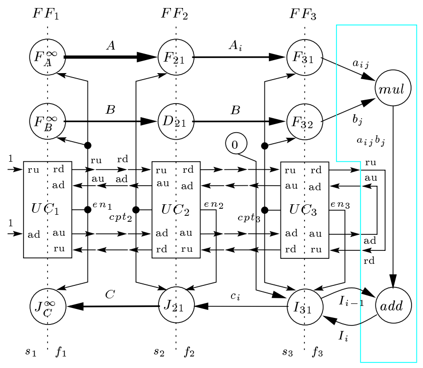

Fig. 6. Implementation graph of MVP

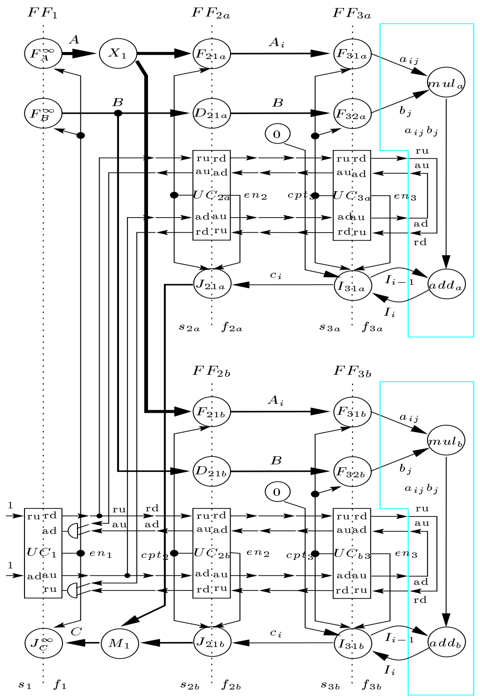

Fig. 7. A defactorized implementation graph of MVP

UC_{3a} and UC_{3b}, The synchronisation of frontiers FF_{2a}, FF_{2b} is assured by the AND gate at the upstream request and the downstream acknowledge of UC_1.

Tab.8 shows the synthesis result of the generated VHDL code of hardware implementation of MVP (6×6 matrix and 6 elements vector, coded on 3 bits) onto a *Xilinx* FPGA XL4000XL-3 4005xIPC84, using the CAD tool *Leonardo Spectrum 2003*, developed by *Exemplar Logic Inc.*. The implementation results are presented in function of, the area (hardware ressources: number of CLBs), the number of clock cycles required by the algorithm execution, the maximum frequency of operators in *MHz*, and finally the latency in *ns*.

7 Conclusion and future work

We have shown that from an algorithm specification based on a factorized data dependence graph model it is possible to generate automatically hardware implementation onto an FPGA circuit, employing a set of rules for the data path and the control path synthesis. The delocalized control approach presented in this paper allows the CAD tools used for the synthesis to place the control units closer to the operators to control rather then a centralized control approach. We

Implementation	*Area (CLB)*	*Nb. cycl.*	*Freq. (MHz)*	*Lat. (ns)*
Factorized Spec.	76	36	12,4	2916
Part.defac. by FF_2	99	18	13,5	1332
Fully. defac. by FF_2	168	6	14,3	420
Part. defac. by FF_3	92	30	10,8	2790
Fully. defac. by FF_3	79	6	9,0	660
Fully. defactorized	234	1	11,4	87

These results represent some possibles implementations explored by the optimization heuristic by partial defactorization (as described in[4]) of the initial factorized implementation. Note that these defactorized solutions allow to reduce the latency of the implementation, but they increase the number of required hardware ressources (CLB).

Fig. 8. Optimization results for the implementation of MVP onto FPGA

validated this design flow on several examples of low-level image processing applications that includes interesting cases of data factorization like: mean filtering [4], edge detector operators (sobel, deriche, ...).

This work is part of the extension of the AAA methodology implemented in the software SynDEx to support implementation on reconfigurable circuits. Basically, AAA/SynDEx for multiprocessors, allows to generate automatically the dead-lock free executive for the optimized implementation of the given algorithm onto the specified multiprocessor architecture [3].

The principles described in this paper allowed us to carry out an automatic generator of structural synthesizable VHDL for mono-FPGA (one FPGA) architectures, that has been added to SynDEx [7]. 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.

Presently we are working on the control involved by the conditioning in the algorithm specification, in addition to the control involved by repetition of operation. We intend to extend the proposed methodology to the case of multi-FPGAs architectures. To support such architectures, the optimization heuristic will adress both defactorization and partitioning issues.

Thanks to this extension, the AAA methodology will be used for optimized hardware/software codesign, leading to the generation of either executives for the programmable parts of the architecture (network of processors), or structural synthesizable VHDL for the non-programmable parts (network of application specific circuits and/or FPGA).

References

1. S. Edwards, L. Lavagno, E.A. Lee, A. Sangiovanni-Vincentelli. *Design of embedded systems: formal models, validation, and synthesis. Proceedings of IEEE*, v.85, n.3, March 1997.
2. C. Lavarenne, Y. Sorel. *Modèle unifié pour la conception conjointe logiciel-matériel.* Traitement du Signal, v. 14, n. 6, 1997, p. 569-577.
3. T. Grandpierre, Y. Sorel, *From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs trans-*

formations. First ACM & IEEE Intl. Conference on formal methods and models for codesign. MEMOCODE'03, june 24th-26th, 2003, mont saint-michel, france.

4. 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.
5. T. Grandpierre, C. Lavarenne, Y. Sorel. *Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors.* CODES'99 7th Intl. Workshop on Hardware/Software Co-Design, Rome, May 1999.
6. D. Gajski, F. Vahid. *Specification and design of embedded hardware-software systems. IEEE Design & Test of Computers*, Spring 1995, p. 53-67.
7. 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.
8. N. Halbwachs. *Synchronous programing of reactive systems.* Kluwer Academic Publishers, Dordrecht Boston, 1993.