Usage-Based Characterization of Complex Functional Blocks for Reuse in Behavioral Synthesis

Nong Fan, Viraphol Chaiyakul

Y Explorations, Inc.
20902 Bake Parkway, Suite 100
Lake Forest, CA 92630
Tel: +1-949-457-0294
Fax: +1-949-457-0437
e-mail: {nfan,viraphol}@yxi.com

Daniel D. Gajski

444 Computer Science Building
University of California
Irvine, CA 92697
Tel: +1-949-824-4155
Fax: +1-949-824-4155
e-mail: gajski@ics.uci.edu

Abstract — This paper presents a novel usage-based characterization method to capture pre-designed complex functional blocks for automatic reuse in behavioral synthesis. We identify attributes necessary for reuse of such complex components and illustrate how the attributes are captured into a design database. A complex component $MTX\_MULT8X8$, which computes product of two 8×8 matrices, is captured with the proposed method, and its reuse in behavioral synthesis is demonstrated with design of a DCT example. Feasibility of the method for capturing various components is demonstrated as well.

I. Introduction

Pre-designed and pre-verified complex functional blocks, such as FFT, FIR and DCT cores, have mask layout data with physical layout information. They provide predictable performance and physical design information since they are tuned to a specific process. Thus, reusing such complex components offers great potential for reducing design time and cost for System-On-Chip (SoC) designs.

To reuse pre-designed components in synthesis, they need to be properly characterized. Characterization is a process in which attributes of a component necessary for its reuse are identified and captured in a design database. Traditionally, attributes such as input timing constraints, output delays, area and power consumptions are characterized and stored in a design database. We refer to these characteristics as **component-based characteristics**. However, component-based characteristics are not enough for automatic reuse of complex components.

Fig. 1 shows an example of pre-designed complex components, $MTX\_MULT8X8$. It performs multiplication of two 8×8 matrices, where each element in the matrix is a 16-bit number, either unsigned ($TC = 0$) or signed ($TC = 1$). Fig. 1(a) shows the interface description of the component. Fig. 1(b) shows the constraint on the minimum clock period and reset pulse. Input matrices are read into it through the 16-bit data input port DIN according to the hand-shaking protocol shown in Fig. 1(c). The resultant matrix is sent out through the 16-bit data output port $DOUT$ according to the hand-shaking protocol shown in Fig. 1(d).

Reusing components such as $MTX\_MULT8X8$ with component-based characteristics and supplemented timing diagrams would proceed at Register Transfer Level (RTL), where designers have to manually instantiate instances of the components in their RTL codes and hand-generate interface control logics. The drawbacks of such an approach are that: (1) reading and understanding documentation have to be repeated every time a component is reused and they are time-consuming; (2) instantiation

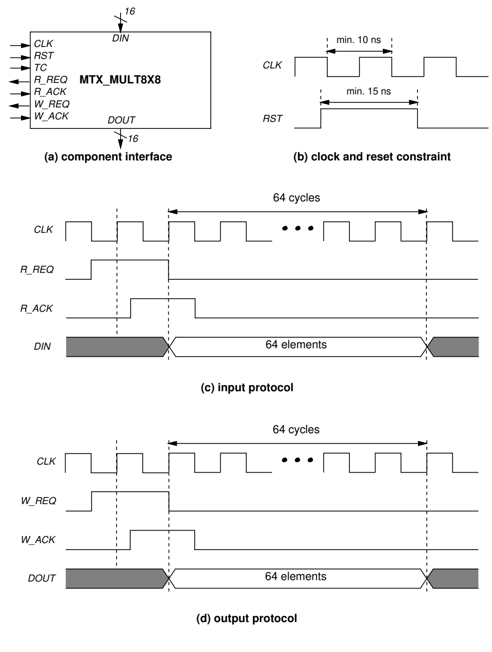

Fig. 1. MTX_MULT8X8: (a) interface description, (b) clock and reset pulse constraint, (c) input protocol, and (d) output protocol.

of components in a design description makes the design difficult to modify since selecting alternative components means that the design has to be rewritten; (3) manually generating the control logics is tedious and error-prone, and (4) the control logics are mixed with the rest of a design and have to be re-verified every time a component is reused.

In order to support automatic reuse in behavioral synthesis, attributes of such components necessary for their automatic reuse must be identified and captured into a design database. In this paper, we propose a novel usage-based characterization method to capture the components such that they can be automatically reused in behavioral synthesis. We will concentrate on what attributes of a component need to be captured and how the information is described in the design database. The usefulness of the captured information in behavioral synthesis is demonstrated by design of a DCT example, and the feasibility of the method for capturing components with various complexities is demonstrated as well.

II. Related Work

In a traditional behavioral synthesis methodology, a designer synthesizes a behavioral description into an RTL net list using a generic library of synthesizable and/or parameterizable components. Each component in the synthesized RTL net list is then synthesized or mapped to an existing component in a target library later. There is a previous work addressing this issue [1]. Libraries used in behavioral synthesis tools, such as BdA [2], Synopsys behavioral compiler [3], OSCAR [4] and CATHEDRAL-III [5], differ in the complexity of their components.

Some recently introduced behavioral synthesis tools, such as CADDY-II [6] and AMICAL [7], increase the complexity of reusable components in their libraries to include components which can perform part of the system specifications.

CADDY-II synthesizes a component from its behavioral specification into an RTL structure consisting of sub-components. Then the information of the component, such as its behavioral specification, the synthesized RTL structure and the determined schedule, are captured into the design database with the goal of sharing the sub-components in future designs.

In AMICAL, components are captured with four different views: (1) the conceptual view specifying operations able to be performed by the component, (2) the behavioral view specifying the operations that can be called from behavioral description, (3) the implementation view specifying the external ports, and (4) the high-level synthesis view linking the behavioral operations and implementations and providing a fixed schedule for the execution of the operations. In order to reuse such a component, a designer needs to manually instantiate it in the input description, meaning that its reuse is not automatically achieved. Both CADDY-II and AMICAL do not consider how to capture a pre-designed complex components with arbitrary I/O protocols and determined timing constraints.

ALOHA [8] proposed a strategy to capture I/O signaling protocols of hardware modules with an "event graph" to support automatic generation of interface between interacting modules from a high-level specification. However, their method does not support automatic component selection during synthesis.

On the other hand, our usage-based characterization provides a method of capturing a pre-designed complex functional block to support behavioral synthesis tasks, such as component selection, scheduling, binding and architecture generations.

III. Problem Definition and Target Architecture

Our problem is defined as follows: given a pre-designed functional blocks, identify attributes necessary for their reuse and capture them into a design database such that behavioral synthesis systems can automatically reuse the components without knowing their implementation details.

In order to support automatic reuse in synthesis, it is desirable to view a component at a sufficient high level of abstraction which describes what operations the component can perform and hides details about how the component performs a particular operation. Meanwhile, information about how to control the component to perform the required operation must be available to synthesis systems. Therefore, the following two kinds of attributes of a component need to be captured:

- functionalities of the component, and
- information about how to control the component to implement a particular functionality.

Behavioral synthesis usually involves allocation, scheduling and binding. Capturing functionalities of a component into a design database supports the task of allocation and binding since synthesis systems can search the database for a capable component based on its functionalities. Capturing control information supports the task of scheduling since the control information contains details about how to control the component to perform a required operation.

Fig. 2 shows the target architecture which allows us to design a circuit using components with various complexities. It is basically a Finite State Machine with Datapath (FSMD) [9]. The FSM controls components in the datapath. In addition to traditional RTL components, the datapath may contain complex components(CC). These complex components may run at their own clocks and communicate with the FSM via complex protocols. Here we call the clock controlling the FSM as the system clock (SCLK) and the clocks controlling datapath components as auxiliary clocks (ACLKs).

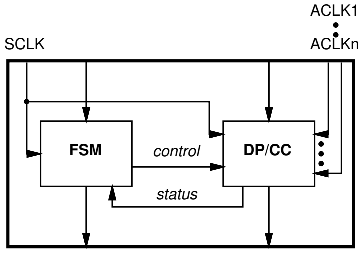

Fig. 2. Target architecture.

IV. Usage-Based Component Characterization

The usage-based component characterization is to characterize a component based on its **usage**, i.e., the functionality of an operation the component can perform and the constraints it has to satisfy. There are two tasks in usage-based characterization, specifying a usage and specifying an interface protocol. The **interface protocol** describes how to perform the operation under the given constraints in the usage. The result of characterization is a set of **binding rules**. Each binding rule relates one usage with its corresponding interface protocol. In order to reduce the number of binding rules for a component, **interface generators** are introduced for usages with flexible constraints.

A. Specifying Usages

The usage describes the functionality of the operations a component can perform and the constraints it has to satisfy to perform the operations.

The functionality of an operation is modeled as a either pre-defined or user-defined function or procedure in a description language by specifying the name of the function/procedure and parameters and their data types. The function/procedure can be invoked in a behavioral description. There are two kinds of constraints in a usage. The constraints on bit width of operands specify the size of the operands the component can take, while the constraints on design clock(s) specify the requirements for both the system clock (SCLK) and auxiliary clocks (ACLKs).

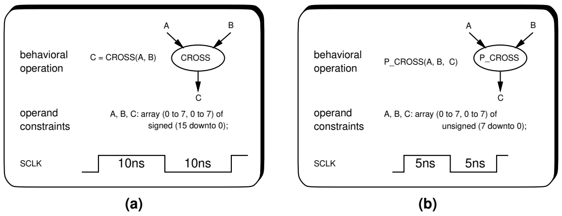

Fig. 3. Two usages for MTX_MULT8X8.

As we mentioned before, the $MTX\_MULT8X8$ can perform multiplication of two 8×8 matrices with each element representing a 16-bit signed number. This functionality can be modeled as a function "$C = CROSS(A, B)$", where A, B and C are 8×8 arrays of 16-bit signed numbers, as shown in the usage in Fig. 3(a). The constraint on the minimum period of the system clock (SCLK) is $20ns$ and it is necessary for design of interface protocol which will be explained later. The $MTX\_MULT8X8$ can also perform multiplication of two 8×8 matrices with elements of 8-bit unsigned numbers. The functionality can also be modeled as a procedure "$P\_CROSS(A, B : in;\ C : out)$", where A, B and C are 8×8 arrays of 8-bit unsigned numbers, as shown in Fig. 3(b). The constraint on the minimum period of the SCLK is $10ns$.

A component may have multiple usages if it can perform different operations or perform the same operation under different constraints. On the other hand, different components may have the same usage since there may exist more than one component in a design database which can perform the same operation under the same constraints. A component would be selected to perform a required operation **if and only if** the functionality of the operation is matched, and the constraints on design clocks and operand bit width are satisfied.

B. Specifying an Interface Protocol for Each Usage

The purpose of an interface protocol is to provide control information necessary for performing a behavioral operation on a component under the given constraints. By comparing the operation "$CROSS$" shown in Fig. 3(a) and the component $MTX\_MULT8X8$ shown in Fig. 1(a), we observe that: (1) they have different number of ports, and (2) the component usually uses more complicated mechanism to receive operands and send results than the operation. Thus, an interface protocol is required to bridge the gap between a behavioral operation and a component.

The control information contained in an interface protocol should specify external control and data flow requirements of the operations performed by the component, such as how to start the operation, how to supply data, read/write protocols, input/output operation timings etc. An interface protocol also needs to guarantee that all timing constraints of the component, such as clock periods, setup and hold timing constraints, are satisfied.

An interface protocol is specified as a state machine, where I/O operations are specified for each state and state transitions are controlled by the system clock (SCLK). Since the period constraint on the SCLK is important for designing the interface state machine, it has to be specified in the corresponding usage of the interface protocol.

An interface protocol is related to a usage by a **binding rule**, which tells that if the usage is matched, the interface protocol can be used to control the component to perform the operation specified in the usage.

A binding rule of the $MTX\_MULT8X8$, which specifies an interface protocol for the usage in Fig. 3(a), is shown in Fig. 4. A control scheme to control the $MTX\_MULT8X8$ to perform the "$CROSS$" is specified

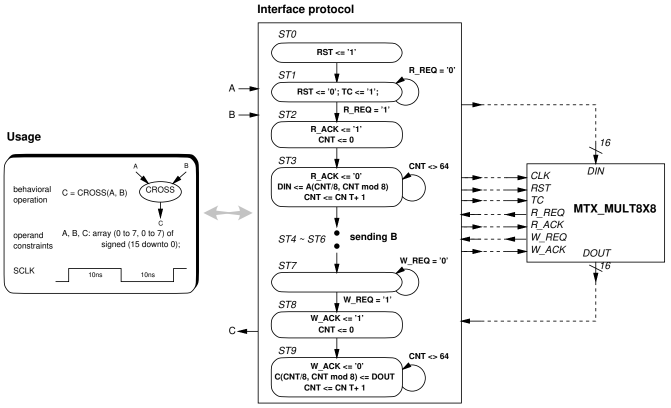

Fig. 4. An example binding rule for the MTX_MULT8X8.

in the interface protocol as a state machine. In the initial state $ST0$, it resets the $MTX\_MULT8X8$ by sending '1' to the port RST. In the state $ST1$, it pulls RST back to '0', sends '1' to TC, indicating that the operands should be treated as signed numbers and waits for R_REQ to become '1'. Once the R_REQ becomes '1', it goes to the state $ST2$ and sends '1' to R_ACK. Then it stays at the state $ST3$ for 64 cycles and sends one element (row wise) to the component at each cycle. Similarly, in the state $ST4$ to $ST6$, it sends the 64 elements of the second matrix B to the component. Then it waits for the computation to finish and fetches the resultant matrix in the state $ST7$ to $ST9$. Since the clock period is no less than $20ns$, the $15ns$ pulse constraint on the RST is satisfied by keeping it high for one cycle.

The interface protocol actually captures design knowledge about how to use a component to perform a particular operation. If the simulation model of the component is available, the interface protocol may be verified by simulating the state machine together with the component and checking whether the resultant matrix generated in the interface is correct. Once the interface protocol is verified and captured into a design database, it may be used directly by a synthesis system to control the component to perform a required operation.

C. Interface Generators

Different constraints on performing an operation usually require different control schemes. For instance, performing the operation specified in Fig. 3(b) under a system clock with the minimum period of $10ns$ requires that the RST signal be kept high for at least two cycles to satisfy the $15ns$ pulse constraints. Furthermore, the 8-bit matrix elements need to be expanded to 16-bit to match the 16-bit input port DIN, and the 8-bit result needs to be extracted from the 16-bit output port $DOUT$ to match the size of the elements of the resultant matrix C. Unfortunately, such constraints are not known until synthesis. In order for a component to be used under various design constraints, it is necessary to specify interfaces suitable to perform an operation under all possible design constraints and store them into the design database. However, such a method will require a huge number of binding rules to be specified for the component, and result in a huge design database. The interfaces determined during characterization are referred to as **static interfaces**.

In order to be able to use $MTX\_MULT8X8$ to perform multiplication of two 8×8 matrices with each element less than 16-bits and representing either unsigned or signed numbers, for instance, we might need to specify $16 \times 16 \times 2 = 512$ binding rules. Considering possible variations of the system clock periods makes this number even larger.

To solve this problem, we introduce usages with flexible constraints and interface generators. The flexible constraints on the operand sizes and design clocks are specified as ranges. For example, the constraint on the element size of the 8×8 matrix would be greater than 0 and less than 17, and the constraint on the minimum system clock period would be greater than $10ns$. An interface generator is a programmable interface protocol for a usage with flexible constraints. It needs to be programmed in such a way that it takes as input the constraints to perform an operation and generates as output a customized interface protocol if the constraints satisfy the flexible constraints specified in the usage.

An interface generator usually consists of an interface template and a set of transformations to manipulate the template to produce a customized interface protocol. The interface template specifies a basic state machine which contains necessary information for generating a customized interface protocol. Since the constraints passed into an interface generator include constraints on size of operands and constraints on design clocks, three kinds of transformations may be applied to the template: state splitting, operand expansion and result extraction.

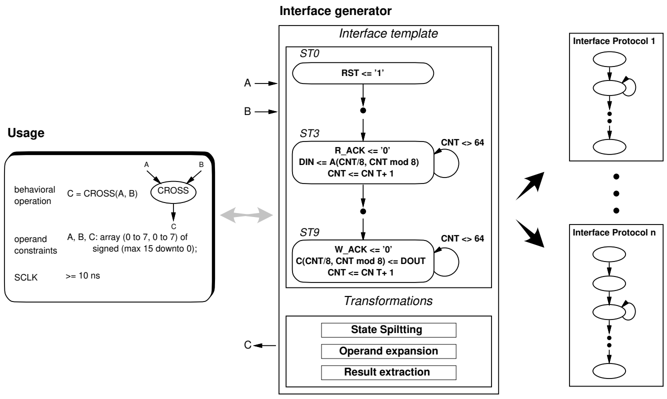

Fig. 5. Interface generator.

The state splitting splits a state in the template into several states with all the statements in the original state duplicated into each post-split state. It is usually used to satisfy constraints on input ports, such as pulse constraints and setup/hold timing constraints, by sustaining the values on the input ports for more than one system clock cycles. The number of states may be determined by dividing the worst case timing constraints among all the signals in the original state by the system clock period. Operand expansion expands operands of a behavioral operation to match the size of component input ports, while result extraction extracts data from component output ports to match the size of the result expected by the behavioral operation.

Fig. 5 shows an interface generator to generate interface protocols to control the $MTX\_MULT8X8$ to perform multiplications of two 8×8 matrices as long as the bit-width of the matrix element is no greater than 16 and the minimum period of the SCLK is no less than $10ns$. The state $ST0$ in the interface template will be split into $\lceil \frac{SCLK}{15} \rceil$ states, where $SCLK$ represents the period of the system clock specified during synthesis and 15 is the minimum pulse width constraint on RST port. The element of the first matrix needs to be expanded in the state $ST3$ and the result is extracted in the state $ST9$. This reduces the number of binding rules to 1. But we can still obtain appropriate control schemes to control the $MTX\_MULT8X8$ to perform the required functionality under various design constraints.

V. Complex Functional Block Reuse in Behavioral Synthesis

Complex functional blocks with usage-based characteristics are able to be automatically reused in behavioral synthesis. With a set of binding rules stored in a design database for a component, the database search engine can use a behavioral operation and design constraints passed by synthesis systems as keys to search for components able to perform the operation under the given constraints. The search results, which are components and their usage-based characteristics, are returned back to the synthesis system. Once a component is selected by the synthesis system, the behavioral description needs to be modified to include the control scheme specified in the interface protocol to control the selected component. A natural approach is to embed it into the original description and use it to control the component. [10] proposed a modified list scheduling algorithm which takes such a control scheme into consideration during scheduling.

Reuse of the $MTX\_MULT8X8$ in synthesis of a 2-dimensional DCT example is demonstrated in Fig. 6. The definition of the DCT for an $8{\times}8$ image can be represented as follows [11]:

$$COEFF \times A \times COEFF^T$$

where A is the input image, $COEFF$ and $COEFF^T$ are DCT constant array and its transpose, respectively.

The input behavioral description mainly contains two function calls to "$CROSS$" and is synthesized under a

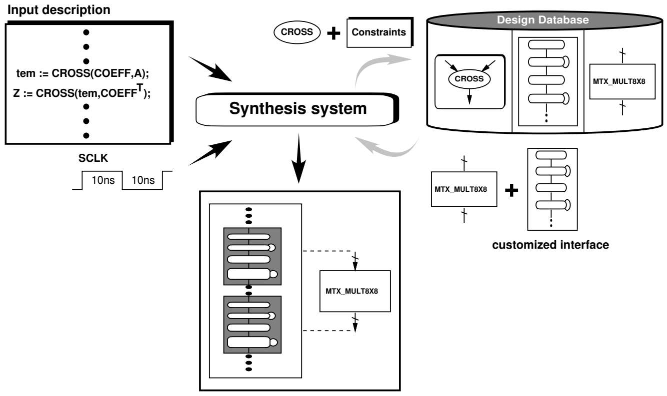

Fig. 6. Synthesis of a DCT example.

system clock with the minimum period $20ns$. The synthesis system needs to query a design database to find components able to perform the "$CROSS$" under the given design constraints. If the $MTX\_MULT8X8$ is selected from the returned components to perform the "$CROSS$", the synthesis system can instantiate an instance of the $MTX\_MULT8X8$ and embed the customized interface protocol into the description. Since there is a data dependence between the two "$CROSS$" operations, they can be performed on the same component. The interface protocol is used twice to control the $MTX\_MULT8X8$ to perform the two operations.

VI. Experimental Results

We have captured a variety of pre-designed complex functional blocks into a design database, eXploration Database (XD)[1] of Y Explorations, Inc. (YXI) using the proposed usage-based characterization method. The components can be automatically reused during synthesis process using YXI's eXploration Compiler (XC)[2].

In the first experiment, we demonstrated that usage-based characterization supports efficient reuse of complex components by designing the DCT example with three different allocations and two design clock constraints as shown in Table I. Complex components, such as shift-and-add multiplier and the $MTX\_MULT8X8$ were used in the designs. The DCT algorithm was described in two ways. One uses nested loops with addition and multiplication operations inside (input 1) and the other uses "$CROSS$" function calls (input 2). Column 2 in Table I lists the number of lines of VHDL codes for both descriptions. Column 3 lists three allocations for the designs and column 4 lists the clock constraints. For comparison, column 5 lists the number of lines of VHDL codes a designer would write to design the DCT with specified allocation and design constraints at RTL. Column 6 lists

[1] XD is a trademark of YXI.
[2] XC is a trademark of YXI.

the increment percentage of code size of these two kinds of descriptions. The last column lists the number of states in the RTL codes.

TABLE I THREE DESIGNS FOR DCT

design	# of lines of VHDL codes to reuse FUs with usage–based characteristics	allocation	clock (ns)	# of lines of VHDL codes to reuse FUs with comp.–based characteristics	% increase	# of states
input 1	61	1 8–bit array mult 1 19–bit adder	20	224	267	28
			10	252	313	32
		1 8–bit SAM 1 19–bit adder	20	278	355	40
			10	298	388	44
input 2	35	1 8–bit 8x8 matrix mult	20	319	811	42
			10	327	834	44

The input 1 and 2 have been synthesized by XC into a structural implementation for each allocation and clock constraint. The database support relieves the designers from understanding I/O protocols and timing diagrams of the components.

In the second experiment, we demonstrated the feasibility of our method for capturing functional blocks with various complexities. Table II lists 7 components we captured using the usage-based characterization method. All the components were designed using MSU SCMOS 0.8 standard cell library [12].

TABLE II USAGE-BASED CHARACTERIZATION FOR 7 COMPONENTS

Component name	Features	Clock reqirements	I / O protocols	Automatic reuse in behavioral synthesis	
				Comp.–based char.	Usage–based char.
16–bit carry lookup adder	combinational FU	0	No	Yes	Yes
16–bit ALU	multi–functional unit	0	No	Yes	Yes
16–bit shift–and–add multiplier	data–dependent execution time	1	customized protocol	No	Yes
16–bit Radix'4 Booth multiplier	two phase clocking	2	one–way hand shaking	No	Yes
16–bit four stage pipelined multiplier	pipelined	1	No	No	Yes
8x8 matrix multiplier	array computation	1	customized protocol	No	Yes
8x8 DCT	array computation	1	customized protocol	No	Yes

The 16-bit adder and 16-bit ALU show that our method can also be applied to capture combinational and/or multi-functional unit. The 16-bit shift-and-add multiplier has data dependent execution time. The 16-bit Radix-4 Booth multiplier is implemented using two phase clocking scheme. The 16-bit pipelined multiplier has four pipe stages. The $MTX\_MULT8X8$ is the component we used in the paper. And the DCT is the design synthesized using the $MTX\_MULT8X8$. All the components with usage-based characteristics can be automatically reused in behavioral synthesis.

VII. CONCLUSION AND FUTURE WORK

We have presented a novel usage-based characterization of pre-designed complex functional blocks for automatic reuse in behavioral synthesis. The I/O protocols and timing constraints of the components are captured into interface protocols based on the operations and the design constraints. The proposed method has been implemented in the eXploration Database (XD) of Y Explorations, Inc.. Experimental results have demonstrated its efficiency and feasibility in reuse of complex components in behavioral synthesis.

REFERENCES

[1] P. Jha and N. Dutt, "Design reuse through high-level mapping," *Proc. of European Design & Test Conference*, 1995.

[2] L. Ramachandran and D. Gajski, "Behavioral design assistant (BdA) user's manual: version 1.0," *University of California, Irvine, Dept. of Information and Computer Science, Technical report*, 94-36.

[3] T. Ly, D. Knapp, R. Miller and D. MacMillen, "Scheduling using behavioral templates," *Proc. of DAC*, 1995.

[4] B. Landwehr, P. Marwedel and R. Domer, "OSCAR: optimum simultaneous scheduling allocation and resource binding based on integer programming," *Proc. of EURO-VHDL*, 1994.

[5] W. Geurts, F. Catthoor and H. De Man, "Quadratic zero-one programming-based synthesis of application-specific data paths," *IEEE Transactions on CAD*, vol. 14, pp. 1-11, 1995.

[6] O. Bringmann and W. Rosenstiel, "Resource sharing in hierarchical synthesis," *International Conference on Computer-Aided Design*, 1997.

[7] P. Kission, H.Ding and A. Jerraya, "VHDL based design methodology for hierarchy and component re-use," *Proc. EURO-VHDL*, 1995.

[8] J. S. Sun and R. W. Brodersen, "Design of system interface modules," *Proc. ICCAD*, 1992.

[9] D. Gajski, N. Dutt, A. Wu and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design,* Kluwer Academic Publishers, 1992.

[10] H. Juan, *Design Methodology and Algorithms for Interactive Behavioral Synthesis,* Ph.D. Dissertation, University of California, Irvine, 1997.

[11] P.M. Embree and B. Kimble, *C Language Algorithms for Digital Signal Processing,* Prentice Hall, 1991.

[12] http://WWW.ERC.MsState.Edu/mpl/scmos/html/release.html