On Design-For-Reusability in Hardware Description Languages

J. Morris Chang and S. Kagan Agun
Department of Computer Science
Illinois Institute of Technology
10 West 31st Street, Chicago, 60616 USA
{chang, agunsal}@charlie.iit.edu

Abstract

The reuse of electronic components can improve the productivity in the system design. However, without careful planing components are rarely designed for reuse. Hardware Description Languages (HDLs) are commonly used to construct from simple hardware to complex ones. The HDLs allow the creating of reusable models, but the reusability of a design does not come with language features alone. It requires design disciplines to reach an efficient reusable design. The reusability issues and the design methodologies to achieve Design-For-Reusability (DFR) are presented. Quantitative analysis of ten VHDL applications, based on the proposed reusable models, is summarized.

1. Introduction

Hardware Description Languages (HDLs) are used to describe hardware for the purpose of simulation, modeling, testing, design, and documentation of digital systems. These languages provide a convenient and compact format for the hierarchical representation of functional and wiring details of digital systems. Some of HDLs have become industrial standard (e.g. IEEE1076 for VHDL).

Modern HDLs support parameterized designs via language constructs. For example, generic parameters can be used to build reusable components. However, reusability does not come magically with language features. This paper calls for a detailed study for the issues in achieving Design-For-Reusability in HDL.

As design teams face increasing device sizes and shrinking time scales, “design-from-scratch” methodologies struggle to cope with the challenge of system-on-a-chip (SoC) design. The reuse of design material offers higher productivity in SoC design. However, it requires careful design process at the initial development stage in order to create reusable models. Thus, reusability guidelines must be provided to the designers in the SoC approach.

However, making a reusable design in a current project can be very different from making a reusable design for general purpose used by others. The additional efforts come from making the core HDL code generic enough to be used in other design environment. These efforts include, making the core code parameterizable and/or configurable, adequately documenting the core, providing adequate simulation and verification environment and synthesis scripts, providing support to the core integration, and providing future maintenance and upgrades to the core.

This paper focuses on the reusability issues in the design of digital systems with VHDL. In VHDL, for example, generic parameters are used to build parametrized component models. The specific behavior of these models is instantiated as the generic parameters are determined by the design entities that use them. This is very similar to the concept of templates in object-oriented language C++.

It is becoming a common practice to offer synthesizable hardware cores which are built with VHDL (a.k.a. *softcore*) from the ASIC vendors. These component models aim to exploit the reusability and the portability of VHDL. As a result, components from the primitive LPMs (Library of Parameterized Modules) to the complex IP (Intellectual Property) are found in recent designs. However, the methodologies to achieve Design-For-Reusability (DFR) are rarely discussed.

This paper attempts to present the design methodologies that may lead to design reuse in using modern HDL. These methodologies are application oriented. For example, the design methodology for a reusable component (*n*-bit adder) which can be instantiated to any precision is considered as precision oriented. Apparently, the objective of such reusable design is to have the design to be used for variable precision. Other design methodologies, such as feature oriented, and data-type oriented are also presented in this paper.

The measurement of reusability has been a major topic in software engineering. The reusability of VHDL codes can be expressed by measuring the lines of code in reusable material or the number of reusable objects. On the hardware design, it is very nature to deal with components. Thus, the number of objects reused is adapted in measuring the reusability of VHDL codes. Ten real world VHDL applications, including popular IEEE packages, are evalu-

ated with proposed design methodologies.

The remainder of this paper is organized as follows. Section 2 presents the current issues of Design-For-Reusability. Section 3 summarizes the previous work. Section 4 details design methodologies for DFR. Section 5 describes measurement issues and shows the results. Last section presents the conclusions of this paper.

2. Current issues in reusable design

The reusability of a design has to do with, at least, the complexity of the design, the design methodologies employed, and the constructs of the HDL used. As the VLSI technology advances continuously, the complexity of hardware components has increased sharply. The complexity of current hardware components can vary from a simple multiplexor to a PCI core (with a 91-page manual). Apparently, the reusability of a PCI core may never come close to the reusability of the MUX in LPM. However, with careful design and a rich libraries of reusable components, the reusability of a complex design can be greatly improved.

Bjarne Stroustrup, the C++ creator, has a phrase about reuse: "Reuse is not something that comes magically from language features; it comes from thinking clearly about issues and getting concepts represented."[1]

With VHDL, the reusability of a design does not come naturally with *generic* construct. It requires substantial design disciplines to come up a reusable design. The disciplines require thorough understanding of both the problem domain and solution domain. In the digital system design, there are typically several approaches to the same problem. However, not all of them can achieve the same reusable objective. To choose an approach that facilitates reusability is not straightforward. This issue in DFR is related to design discipline. The discipline can be built with the help of a set of well defined application oriented design methodologies. Therefore, we investigate the design methodologies based on different reusable orientation. Hoping that these design methodologies can help us to develop the basic discipline to achieve DFR.

Apparently, DFR can't be done without the supporting features from HDL. In VHDL, these features include *generic, generate* and *procedure*... etc. However, being a strong type language will yield less flexibility in the design. For instance, without careful design, a binary adder may accept only the operands of certain data type. This way, the reusability of the adder is limited. Sometime, it is desirable to have a reusable component that can work with input/output of different data type. These issues in DFR are related to language constructs.

3. Previous work

Generic component library was one of the first ideas to create customized component libraries. Customized VHDL component libraries would provide platform independence and increase the reuse of components in the library. LEGEND[2][3], generator-generator language has been introduced to generate generic component libraries. LEGEND complements VHDL by providing a library generator based on hierarchy and inheritance. SUAVE [4], a language extension to VHDL, extends the genericity mechanism by providing formal generic types. It promises that the units will be reused in a much wider area without modifying code. SUAVE adds the type feature to VHDL to provide generic components such as integer and floating point multiplexers. Today, most of modern HLDs already provide this extension.

A reusable component model [5] has been proposed to maximize the reusability. This approach is based on hierarchical design methodology in the design phase which enables the reusable knowledge with Hierarchical Object Oriented Design (HOOD) method. A class hierarchy based on object oriented class methodology [6] and abstract design levels[7] are the similar approaches for reusable component design. These approaches can't provide methodologies for reusable component design.

Coding techniques[8] presented reusable examples through language features. These features are not enough to achieve reusability. Another approach, component-base design technique which divides functionality of a design into internal functional behavior and external interfacing [9] requires a special compiler and interface libraries. Thus, the reusability of components depend on the platform. Apparently, the key factor for reuse is the employed design methodologies. However, it has rarely been reported. Next section details design methodologies for DFR.

4. Design methodologies for DFR

Reusability is based on the orientation of the design. Precision, feature, and data type orientations are the main factors to construct reusable hardware designs. The hardware is reusable when it supports any size of precision. Feature orientation is another factor for reusability. Having an option to choose any feature of a hardware gives us a flexibility without designing separate hardware for each feature. Data type is crucial in reusable hardware design. Data-type orientation provides one component with all possible data type for reusability. A design may include one or a combination of these orientations. For example, a design may use both of precision oriented and data conversion approaches. Design methodologies will be discussed in the following sections to provide detailed information.

4.1. Precision-oriented design methodology

The aim of a precision oriented design is to have a

parameterized design for a digital component. This component can be instantiated to any precision. Typically, the precision refers the number of bits. Thus, this component can be reused regardless of the precision. To achieve this goal, two approaches are presented— bit-slice and non-bit-slice.

4.1.1. Bit-sliced approach

Partitioning a complex design into small components allows engineers to focus on higher level of abstraction and hierarchy in design description which have become desirable to digital system designers. For instance, an 8-bit ripple carry adder can be implemented by a cascade of eight full-adder stages. These full-adder components are instantiated in an iterative fashion. With these components, an 8-bit adder is formed in a structural model. This is referred to bit-sliced with structural model.

On the other hand, an 8-bit adder can be implemented without using components. Instead of using a full-adder component, the data flow model of a full adder is reused in an iterative fashion. This approach is referred as bit-sliced with dataflow model. Next subsections details bit-sliced approaches with structural and data flow models.

Bit-sliced with structural model

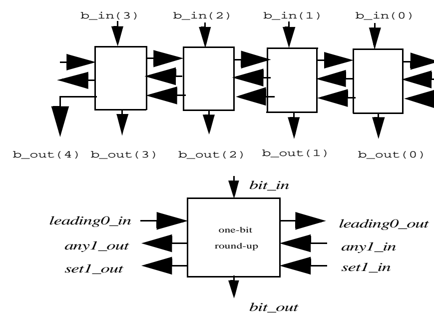
The VHDL **generate** statement can instantiate small components (e.g. 1-bit full adder) in an *iterative* fashion. In addition, using the VHDL **generic** declaration this 8-bit adder can be instantiated to a *n*-bit adder. Such approach results in a reusable design.

However, some of the cases can be less intuitive than the adder. One of our favorite examples in bit-slice approach is a *round-up* system. This system should round up a given *n*-bit binary data (as input) to its next higher binary data (as output) which is 2^N. For example (in decimal), inputs of 0, 2, 5, and 12 will produce outputs of 0, 2, 8, and 16 respectively.

```
b_out(0)=b_in(3)'•b_in(2)'•b_in(1)'•b_in(0)
b_out(1)=b_in(3)'•b_in(2)'•b_in(1)•b_in(0)'
b_out(2)=b_in(3)'•b_in(2)•b_in(1)'•b_in(0)'•b_in(3)'
         •b_in(2)'•b_in(1)•b_in(0)
b_out(3)=b_in(3)•b_in(2)'•b_in(1)'•b_in(0)'
         +b_in(3)'•b_in(2)•(b_in(1)+b_in(0))
b_out(4)=b_in(3)•(b_in(2)+b_in(1)+b_in(0))
```

Figure 1. A Boolean expression approach for a 4-bit round-up system

A non-reusable design based on the Boolean expression is presented in the figure 1. A bit-slice approach for a 4-bit round up system is presented in Figure 2. The bit-slice approach is very similar to the ripple adder. It starts with a careful planning to partition the given task down to single bit level. Then, structural architecture is used to take advantage of VHDL constructs — *generate* and *generic*. Moreover, this design employs a "message passing" methodology to link four one-bit roundup blocks. It is worth noting that the messages are passed in both directions while the *carry* is passed in only one direction in the ripple adder. The discipline of system level partition and algorithm development is crucial to this bit-slice approach.



leading0_in: all the leading bits are "0s"
any1_in: at least one of the trailing bits is "1"
set1_out: the current bit is requesting the higher order bit to set the *bit_out* to "1"

Figure 2. A 4-bit round-up system.

Other examples of using bit-sliced approach include digital comparators in [11]. It seems that the bit-slice approach can be slow due to the signal propagation between each block. However, the actual performance depends on the logic synthesis.

Bit-sliced with data flow model

To illustrate the design approaches using bit-sliced with data flow model, an eight-bit ripple carry adder is used. Instead of using a full-adder component, the data flow model of a full adder is reused in an iterative fashion (Figure 3). shows the VHDL code of this approach. However, in order to make designs synthesizable in some synthesis tools, the bounds of *for* loops must be locally static expressions.

```
for I in 0 to L_LEFT loop
   RESULT(I) := CBIT xor XL(I) xor XR(I);
   CBIT := (CBIT and XL(I)) or (CBIT and XR(I)) or (XL(I) and XR(I));
end loop;
                                    (a)
c(0)<=cinn;
for i in 0 to N-1 loop
   sumn(i) <= an(i) XOR bn(i) XOR c(i);
    c(i+1) <= (an(i) AND bn(i)) OR (an(i) AND c(i)) OR (bn(i) AND
c(i));
end loop;
coutn <= c(N);
                                    (b)
```

Figure 3. Implementation of bit-sliced with data flow model

4.1.2. Non-bit-sliced approach

```
shift: PROCESS (data_in, r_l_shift, c_tmp)
BEGIN
    data(0) <= data_in;
    FOR i IN 0 TO n-1 LOOP
        IF i < c_tmp AND r_l_shift = '1' THEN-- shift right
            data(i+1) <= data(i)(0) & data(i)(n-1 DOWNTO 1);
        ELSIF i < c_tmp AND r_l_shift = '0' THEN-- shift left
            data(i+1) <= data(i)(n-2 DOWNTO 0) & data(i)(n-1);
        ELSE
            data(i+1) <= data(i);
        END IF;
    END LOOP;
    data_out <= data(n);
END PROCESS;
```

Figure 4. N-bit barrel shifter

Non-bit-sliced approach is also used in precision oriented design. Behavioral description of the designs can be instantiated to any precision through parameters. Figure 4 presents an *n*-bit barrel shifter that uses a *for-loop* construct. Parameter n in for-loop determines the precision of

the barrel shifter.

4.2. Feature-oriented design methodology

The aim of feature oriented design is to have a reusable component that can offer wide range of features in a digital system. For example, the *LPM_add_sub* component can offer either addition or subtraction function (in signed or unsigned) to the given operands. It is convenient to implement related features into one design so that this design can be reused again and again as different feature is required. We classify this design methodology into three subgroups based on the relationship of features.

One technique used in object-oriented analysis is to discover the relationship of classes. The most common relationships are “is-a” and “has-a”. For example, a shift register “is a” register. A multiplier-accumulate (MAC) unit “has a” multiplier. It is also very common to mix these two relationships into one design; we refer it as a “hybrid” relationship. The next diagram presents these relationships.

In Figure 5, we show feature-oriented design methodologies that use *case-when* construct in VHDL to select the desired feature(s). In the *case a*, the available features to the base model *A* are *B* and *C*. Thus, the reusable component can be instantiated to one of four different designs— model *A* without any extra feature, with feature *A*, with feature *B*, or with both feature *A* and *B*. In the *case b*, *A* and *B* are two independent features. Thus, the reusable design can be instantiated to one of three different designs— with feature *A*, with feature *B* or with both features.

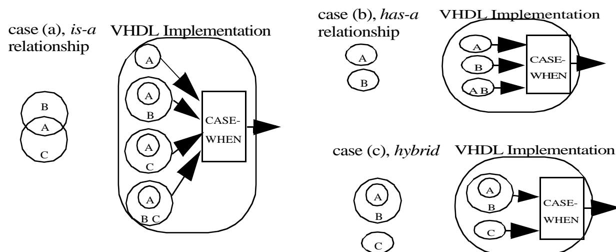


Figure 5. Relationship of components and their VHDL implementations

Recently, language constructs to support inheritance between VHDL entities are proposed [7]. These language constructs can facilitate the implementation of “is-a” relationship. However, without these constructs, one still can design a reusable component through case-when construct. To our knowledge, the “has-a” relationship is more common than “is-a” relationship in the digital design. This is very different from the software systems.

4.3. Data-type-oriented design methodology

It is desirable to have a functional unit to accept operands of different data type. However, VHDL is a strong type language that doesn’t provide automatic type casting. Therefore, a user defined type conversion is used for this purpose. This approach add reusability to a hardware component. In the next figure, the *n*-bit adder accepts the operands of *bit_vector* type only. To accept other data type, two type conversion functions are added — one to convert the inputs to bit_vector and one to convert the bit_vector back to desired outputs.

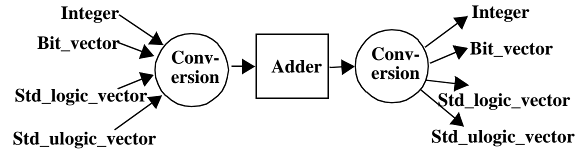


Figure 6. Data type oriented adder diagram

The approach in the above figure put all the type conversion functions into one single place is referred to *centralized data conversion.* Another approach, *distributed data conversion*, provides data conversion in several places. Both of the approaches use the function overloading feature to implement type conversions.

1. *Centralized Data Conversion:* A centralized function will support all the possible data conversions. Figure 6. shows the centralized approach.

Although the conversion functions have no hardware significance to the design, it is crucial to the reusability of a design. It allows different data types to be used in one reusable design. These conversion functions can be implemented with *case-when* construct and a set of overloaded *procedures* and *operators*. A simple example of such implementation is given in next figure.

```
CASE OpType IS                     -- Statement # --
     WHEN IntType=>
        Conv2Bit(N, AInt, aSig);      -- 1 --
        Conv2Bit(N, BInt, bSig);      -- 2 --
        Conv2Int(carry(N),Cint);      -- 3 --
        Conv2Int(N, sSig, Sint);      -- 4 --
        ...
     WHEN BitType =>
        ...
     WHEN StdType =>
        Conv2Bit(AStd, aSig);         -- 5 --
        Conv2Bit(BStd, bSig);         -- 6 --
        Conv2Std(carry(N), CStd);     -- 7 --
        Conv2Std(sSig, SStd);         -- 8 --
        ...
     WHEN StdUType =>
        Conv2Bit(AStdU, aSig);        -- 9 --
        Conv2Bit(BStdU, bSig);        --10 --
        Conv2StdU(carry(N), CStdU);   --11 --
        Conv2StdU(sSig, SStdU);       --12 --
        ...
END CASE;
```

Figure 7. Data-type-oriented adder implementation

In the above figure, *case-when* construct is used to select the desired type conversion function. The actual type conversion functions can be implemented through overloaded *procedures*. The invocations of these procedures are *Conv2Bit()* (in statement #1 & #2) and *Conv2Std()* (in statement # 7 & #8). The implementation of these procedures are placed in *package* construct. Thus, they can be reused in other designs.

2. *Distributed Data Conversion*: For example, components (e.g. adders), with same functionality, are constructed

differently to accept different data types. In this approach, one or more data type conversion functions are integrated into the components. Operator or function overloading are commonly employed.

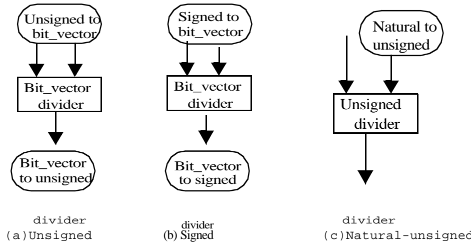


Figure 8. Distributed data-conversion approach.

Figure 8. shows some of the examples of distributed data-conversion. *Unsigned* data inputs are converted into *bit_vector* in order to be divided through a *bit-vector-divider* (Figure 8.a). The result of the division is converted back to unsigned. *Signed-divider* is the signed version of the same component (Figure 8.b). It is also possible to form different dividers which accept different data types. For example, *natural-unsigned-divider* in IEEE numeric-std package has unsigned and natural data type inputs. This divider converts natural data type to unsigned data type in order to use *unsigned-divider* rather than *bit-vector-divider* (Figure 8.c). It is recognized that data conversion increases the reusability of core components.

3. *Other Data Conversions*: It is worth noting that some data type conversions are available from VHDL through type casting. For example, a signed variable *b* can be converted to unsigned data type by using type casting, UNSIGNED(*b*). This is referred to *build-in data conversion*, since this conversion is part of the language features. However, this only works between a data type and its subtypes.

The *std_logic_arith* package in the IEEE library includes four sets of functions to convert values among SIGNED and UNSIGNED types, std_logic_vector, and the predefined type INTEGER. These type conversion functions are: *conv_integer*, *conv_unsigned*, *conv_signed* and *conv_std_logic_vector*. Four versions of each function are available; the correct version for each function call is determined through function overloading.

In the IEEE VHDL synthesis package (IEEE 1076.3, numeric_std, 1995), function and operator overlodings are widely used to facilitate different data types. For example, arithmetic operators (e.g. “+”, “-”, “/”, “rem”, and “mod”) are overloaded to accept operands of different data type. However the data types of the results of these arithmetic operators are limited to either signed or unsigned. Apparently, the conversion functions in the *std_logic_arith* package can be used to covert the results to integer or std_logic_vector. The design approaches of these IEEE packages fall into the data-type-oriented design methodology.

5. Results

Measuring reusability is one of the important topics in software engineering. The most common ways to measure reusability are the line of code (LOC) of the reused objects and the number of the reused objects. On the hardware design, it is very nature to deal with components. Ten real world VHDL applications, including popular IEEE packages, are evaluated with proposed design methodologies. Next Table summarizes these applications.

Table 1. Summary of packages and applications

Packages/Applications	Source	LOC*
IEEE Numeric_bit (ver:2.4) 1995	www.edit.org	1503
IEEE Numeric_std (ver:2.4) 1995	www.edif.org	2201
LPM (ver:1.3) 1997	www.altera.com	1564
Package Math_Real (ver:0.7) 1993 Real Complex	rassp.scra.org	(837) 541 296
Behavior DLX Processor (1993)	rassp.scra.org	1204
8051 Microcontroller (1999)	www.cs.ucr.edu/~dalton/8051	5763
PIC-16C5X Microcontroller (1998)	http://tech-www.informatik.uni-hamburg.de/vhdl/vhdl.html	450
16M SDRAM w/ test bench (1998)	micron.com	6251
DES encryption/descryption (1999)	www.free-ip.com	989

*LOC: Line of Code (excluding comment lines)

In Table 1, these applications include IEEE numeric packages, LPM, and models of microcontroller. These applications represent real world examples. The URL and size of the program (i.e. LOC) are also given in the table.

To analyze these designs, two programs, *measure* and *count,* were developed. *Measure* accepts VHDL codes as input, searches components, and counts LOC and number of reused components in the design. LOC is used to get an idea of the size of the applications. This program generates number of the reused components and the detailed information of these reused component (e.g. function, procedure, component, etc.,) The type of the design methodology used in each reused component is also included in the output file which will be fed to the program *count*. *Count* simply generates the statistics of design methodologies for each application.

Table 2. The summary of design methodology analysis

Package/Library	Number of Functions	adhoc%	Bit-Sliced (structural)%	Bit-Sliced (data flow)%	Non-Bit-Sliced%	Data Conversion%	Feature Oriented%
Numeric_bit	130	4.6	0	2.3	90	3.1	0
Numeric_std	135	2.9	0	2.2	91.9	3	0
LPM	27	3.7	0	11.1	40.7	3.7	40.7
Real	32	100	0	0	0	0	0
Complex	43	90.7	0	0	0	9.3	0
DLX Processor	35	2.9	0	22.9	62.8	11.4	0
i8051 Controller	38	84.2	0	0	0	15.8	0
PIC-16C5X	16	100	0	0	0	0	0
SDRAM*	8	0	0	0	50	50	0
DES encryption	16	100	0	0	0	0	0

* noeprims library functions (16) aren’t included

The results of the distribution analysis are summarized in Table 2. Each function of a given application is examined according to the proposed design methodologies. The functions that use adhoc approach in the design are classified as adhoc. Such functions may be reused many times within the same application. However, to use these adhoc type functions, the parameters must be used in a specific way. This leads to no flexibility to the function. Thus, it has very limited reusability in other applications

Four of the ten applications have a high percentage functions which use adhoc design approach. In the other six applications, the most common design approaches are precision oriented. Among the precision-oriented methodologies, non-bit-sliced is the most common one. Data-conversion design approach seems to play an important role in system design. 60% of IEEE numeric library functions include data-conversion functions. Table 3 presents the detail distribution between distributed and casting data conversions.

Table 3. Distribution of the data conversion functions in non-bit-sliced functions of IEEE numeric_bit and numeric_std packages

Package/ Library	Non-Bit-Sliced	
	Dist. Data Conv.	Casting Data Conv.
Numeric_bit	36.9%	25.5%
Numeric_std	35.5%	24.5%

Among the packages and applications, LPM is the only one which includes feature-oriented design methodologies. It is worth noting that the 40.7% of the LPM implementation are feature-oriented approach. Within that 40.7%, *is-a* relationship and *has-a* relationship are 29.6% and 11.1%, respectively. *If-else* is the common construct to choose proper functionality (e.g. pipelined or non-pipelined) of the components. *Lpm_add*, *lpm_compare*, *lpm_mux* have signed and unsigned versions of a function. Thus, LPM[10] presents feature-oriented reusable design. However, it can handle only one data type— *std_logic_vector*. None of the LPMs can accept more than one data type. Moreover, if needed, the IEEE *std_logic_arith* package mentioned earlier can be used to perform type conversions. In some cases, LPMs accept inputs in the type of string. A conversion function that converts string back to integer is invoked internally.

6. Conclusions

System design requires well defined design methodologies to achieve reusability. This paper presents methodologies that can achieve Design-For-Reusability HDL design at RTL level. Their orientations are based on precision, feature and data type. Detailed implementation of these methodologies is presented.We argue that reuse does not come magically from language features; it comes from design discipline.

Ten real world applications are collected and analyzed in this research to support design methodologies. Applications including IEEE packages and the industrial standard—LPM, are examined with the proposed classification. Adhoc design approach is still the most common one. These functions may have very limited reusability in the future design. The second common design approach is precision oriented. Moreover, the design methodology that focus on data conversion is also very popular in many applications. This indicates that adopting some basic reusability guidelines for the initial development of all code can increase the reusability of this code in the future.

7. Reference

[1] B. Stroustrup, "The Design of C++" recorded 3/2/94, UVC tape.

[2] N. D. Dutt, “LEGEND: A Language for Generic Component Library Description”, International Conference on Computer Languages, New Orleans, LA, pp. 198-207, March 12-15, 1990.

[3] N. D. Dutt, “Generic Component Library Characterization for High Level Synthesis”, Proceedings of the Fourth CSI/IEEE International Symposium on VLSI Design, Los Alamitos, CA, pp. 5-10, Jan. 4-8, 1991.

[4] P. J. Ashenden, P. A. Wilsey, D. E. Martin, “Reuse Through Generic in SUAVE”, Proceedings of the VIUF Conference, Rapid Systems Prototyping with VHDL, Arlington, Virginia, pp. 170-6, Oct. 19-22, 1997.

[5] M D’Alessandro, P. L. Iachini, A. Martelli, “The Generic Reusable Component: an Approach to Reuse Hierarhical OO Design” , Software Reusability, Proceedings Advanced in Software Reuse, Los Alamitos, CA, pp. 39-46, 1993.

[6] J. Böttger, K. Agsteiner, D. Monjau, S. Schulze, “ An Object - Oriented Model for Specification, Prototyping, Implementation and Reuse”, Design Automation and Test in Europe (DATE), Paris, France, Feb 23-26, pp. 303-310, 1998.

[7] W. Röming, M. Radetzki, W. Nebel, “Objective VHDL: Hardware Reuse by Means of Object-Oriented Modeling”, 1st Workshop on Reuse Techniques for VLSI Design, Karlsruhe, Sept. 1997.

[8] S. Meiyappan, K. Jaramillo, P. Chambers, “VHDL Coding Styles for Reusable, Synthesizable Designs”, SNUG’99, Boston, Oct. 7-8, 1999

[9] A. Melikian, D. Altas, G. Fayad, K. Khordoc, “A High Level Aynthesis Approach to Soft IP Reuse”, SNUG’99, Boston, Oct. 7-8, 1999.

[10] “http://www.edif.org/lpmweb/more/210model.vhd”, EDIF web site for LPM sources, 1998.

[11] Z. Navabi, “*VHDL: Analysis and Modeling of Digital Systems”*, 1993, McGraw-Hill.