Reuse Design and Test using Object-Oriented Hierarchical Models Libraries

Fabrice Bernardi, Jean-François Santucci

Reuse Design and Test using Object-Oriented Hierarchical Models Libraries

Fabrice Bernardi **Jean-François Santucci**
University of Corsica, SPE Laboratory
Quartier Grossetti, BP 52, 20250 Corte, France
{bernardi, santucci}@univ-corse.fr

ABSTRACT

We introduce in this paper an object-oriented VHDL Design and Test library in which are saved all the descriptions and testbenches developed during the Design and Validation phases. The design of complex manufactured systems is a task requiring a lot of time to be achieved. One way to speed up this task is to develop methodologies for creating highly reusable components and assisting the reuse process in the design and test phases. A set of recent works has recently dealt with reusable concepts and libraries in the VHDL Design and Test area. The originality of our approach lies in the facts that it is based on a strong notion of genericity of use, and on notions like the inheritance and abstraction links between the stored descriptions. We describe how managing inheritance between stored models can improve their classification and their accuracy, and how the abstraction hierarchy allows to describe a same model at various detail levels.

Keywords: Design, Test, Library, Object-Oriented, Reuse, Hierarchical

INTRODUCTION

The design of complex manufactured systems is a task requiring a lot of time to be achieved. One way to speed up this task is to develop methodologies for creating highly reusable components and assisting the reuse process in the design and test phases. Usually, design tools are associated with libraries of reusable modeling components [2, 3, 12]. Storing models in a common generic library has several benefits. First, the genericity of this storage service can be offered to various modeling and simulation environments. Second, a common library allows environments to share information so they can interact each other, and third, modeling components can be shared by several users. This last point is the most important since it allows a design team enabling an efficient collaborative work.

A set of recent works has recently dealt with reusable concepts and libraries in the VHDL Design and Test area. The definition of the basic elements (VHDL descriptions and testbenches) is a very important part of reuse oriented Design and Test process. The elements are normally defined in terms of concepts which are specific to the class of elements or testbenches for which the model is used. These definitions are generally stored in a library. When an element occurs in a VHDL or testbench description, its definition is copied from the library and placed in the context of the model. Thus an element once defined and placed in the library, can be used as often as required without having to be redefined.

To facilitate the reusability of components, management systems of libraries should provide a nice way to store components and testbenches in order to facilitate the access to components and testbenches, to offer a classification allowing to avoid a memory loss and to propose an elegant way to put stored elements in the context of a particular design. The problems derived from the previous requirements are (1) How to integrate the different levels of abstraction involved in VHDL design (Algorithmic level, RT Level, Gate level, Transistor level) in the framework of the same library ? (2) How to efficiently associate testbenches to VHDL descriptions ? (3) How to provide inheritance between VHDL descriptions or testbenches ? (4) How to facilitate the retrieval of descriptions and testbenches ?

We introduce in this paper an object-oriented VHDL Design and Test library in which are saved all the descriptions (which can be seen as models) and testbenches developed during the Design and Validation phases. This library contains the components which may be used in VHDL descriptions at different levels of abstraction. A full description of these components can be specified just once by a designer and stored in the library. The basic strategy is to provide the four distinct levels of abstraction of a same component with the library framework. Furthermore the designer may define appropriate intermediate models allowing to provide an inheritance between models. As elements must be defined according to two kinds of domains (VHDL descriptions or

 ISBN: 1-5655-283-0

Testbenches) and for each level of abstraction, it follows that separate domains must also be built for each type of elements and level of abstraction. The proposed library is based on a hierarchical organization of concepts. The leaves of the hierarchical structure represent the elements which can be used in model descriptions. Information concerning the elements can be placed at the appropriate level in the hierarchy, thus, eliminating needless repetition when the elements are being defined.

The originality of our approach lies in the facts that it is based on a strong notion of genericity of use, and notions like the inheritance and abstraction links between the stored models. We describe how managing inheritance between stored models can improve their classification and their accuracy, and how the abstraction hierarchy allows to describe a same model at various detail levels.

The paper is organized as follows. Section 2 presents the basics concepts of the VHDL Design and Test. Section 3 introduces the basic concepts of the object oriented approach for defining a hierarchical library. Section 4 presents the architecture we defined for a VHDL Design and Test library. Section 5 presents our web-based implementation of the library. Finally, section 6 concludes this paper and provides some perspectives of work.

ABSTRACTION HIERARCHY AND HDL DESIGN FOR TEST

One of the most difficult tasks in the field of design of complex systems is to choose a good level of detail. In all scientific domains, models are built at a precise abstraction level. The abstraction level of a model determines the amount of information that is contained in the model. The quantity of information in a model decreases with the abstraction levels: a model described at a low abstraction level will contain more information than a model described at a higher abstraction level (Figure 1).

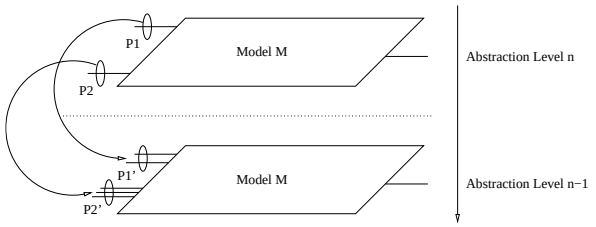

Figure 1: Abstraction hierarchy.

P. Benjamin claims that “determining the correct abstraction level refers to selecting the quantum of information that must be included in the model to help address the modeling goals” [4]. So, well defining the abstraction level is an important step in design. A model described according to several abstraction levels is said a “hierarchical model”. Digital designers employ a set of abstraction levels:

- System level: the system is described using a natural language;
- Algorithmic level (also called System level): the system is described using natural language and an algorithmic description (like C language);
- RTL (Register Transfer Level): explicit definition of all state-holding elements;
- Gate level: all high level constructs (if, case,...) are converted to gates;
- Transistor level: all are converted with nmos, pmos,...
- Layout/Silicon level: final picture of the circuit.

The Silicon level is the lowest level in the hierarchy, the System level the highest. One can represent a design at any of these levels.

VHDL is an Hardware Description Language (HDL, [1, 10]) and can be defined as a high-level programming language with specialized constructs for modeling hardware. It is used to provide the description in the Algorithmic, RTL, Gate and Transistor levels.

An HDL testbench is a program that describes simulation input using standard HDL language procedures. Simply speaking, the test bench is a top level hierarchical model which instantiates the Unit Under Test (UUT) and drives it with a set of test vectors and compares the generated results with expected responses. A typical VHDL or Verilog test bench is composed of three main elements: a Stimulus Generator, driving the UUT with certain signal conditions (correct and incorrect transactions, minimum and maximum delays, fault conditions, etc.), an Unit Under Test (UUT), representing the model undergoing verification and a Verifier, automatically checking and reporting any errors encountered during the simulation run. It also compares model responses with the expected results.

OBJECT-ORIENTED HIERARCHICAL MODELS LIBRARIES

Object-Oriented Architecture of the Models Library

We define, in order to be as precise as possible, two notions: “context-in” and “context-out” models. A context-out model

is an abstraction of a model. It presents a structure allowing it to be stored in a models library. A context-in model is a context-out model extracted from a library and formatted so as to be directly reusable in its environment.

Building a models library is as creating a high level representation of the models and their relations. In order to meet the requirements stated above, we define five objects likely to be stored in a Library object. First of all, a Domain corresponds to the theoretical domain of the stored models (Ex: DEVS Simulation, High Level Synthesis, VHDL Test). Inside Domains, an Application Domain corresponds to the application domain of the considered models (Ex: Microelectronic, Energetic). A Classification Intermediate Model (CIM) belongs to an Application Domain and allows creating a classification hierarchy between the storage objects. This kind of model is not a storage model. An Inheritance Intermediate Model (IIM) is a storage model allowing the share of characteristics with its children through an inheritance mechanism. Finally, a Model File represents a context-out model. It is the basic storage element of the library. We can note that the documentation of the model is included as an attribute of this object.

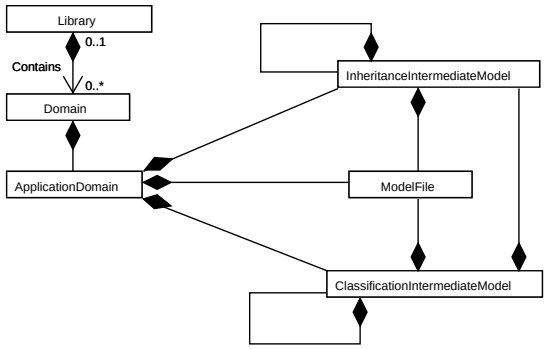

Figure 2: Links between elements.

Figure 2 presents the relations existing between all these objects. We can note that we defined two types of storage objects (Model Files and Inheritance Intermediate Models). An Inheritance Intermediate Model can not contain directly a Classification Intermediate Model, since its children must be a storage object (Model Files or other Inheritance Intermediate Models) for the sharing of properties using the inheritance mechanism.

All these objects are associated in a library with a key. As in a relational database, this key is unique in all the storage space and is used in order to allow the definition of links between the various stored objects.

Abstraction Hierarchy

We defined previously the abstraction level of a model as a level of details. We saw also that a model described at a low abstraction level presents more informations than a same model described at a higher level. In our case, only model files can be associated with an abstraction level. In order to manage this hierarchy, we introduce the notion of *abstraction matrix*. The values of this square matrix are composed by the difference between the two abstraction levels of two models. If models are the same, the value is "0", the same as if they have no "abstraction relationship".

Let $\Omega_{\mathcal{L}_i}$ be the abstraction matrix associated to the library $\mathcal{L}_i$, and let $\omega_{j,k}$ be one of its elements. If $mf_{i,j}$ and $mf_{i,k}$ are two model files, and if $n_{i,j}$ and $n_{i,k}$ are their respective abstraction levels,we have:

$$\Omega_{\mathcal{L}_i} = \begin{pmatrix} 0 & \cdots & \omega_{1,k} & \cdots & \cdots & \omega_{N_{MF},i} \\ \vdots & 0 & \vdots & & & \vdots \\ \vdots & & 0 & \omega_{k,j} & & \vdots \\ \omega_{j,1} & \cdots & \omega_{j,k} & 0 & & \vdots \\ \vdots & & & & 0 & \vdots \\ \omega_{N_{MF},i} & \cdots & \cdots & \cdots & \cdots & 0 \end{pmatrix}$$

$$\text{with : } \omega_{j,k} = \begin{cases} 0, \text{ if } j = k \\ 0, \text{ if } mf_{i,j} \text{ and } mf_{i,k} \text{ are not linked} \\ n_{i,j} - n_{i,k}, \text{ if } mf_{i,j} \text{ and } mf_{i,k} \text{ are linked} \end{cases}$$

We can also write: $\omega_{j,k} = -\omega_{k,j}$.

Moreover, we define the following relation: $\forall i, j, k$, we have $\omega_{i,j} \neq \omega_{i,k}$ or $\omega_{i,j} = \omega_{i,k} = 0$. That means that a same value different from "0" can not appear two times on a same line of the matrix. Thus, an abstraction level defines an unique relationship between two models.

Storage Independence Management

One of the most important objectives of a models library, as defined in this paper, is to be independent from the storage mode. That means that a library must be able to store fundamentally different models coming from fundamentally different modelling and simulation environments. This independence implies that we must dissociate the contents from the format of the stored models, and also that the communication interfaces of a library with the external applications must be identical for all kinds of models.

The storage independence of the library is performed using a "Domain Parser". We call Domain Parser an object able to be used in two distinct modes, and able to analyze or to create a file that describes a context-in model. A Domain Parser relies upon a separation methodology of the extent of the model

from its description format. Thus, the selected approach consists in defining a separation methodology for each domain in a library. Using such a methodology, a Domain Parser allows the user to transform a context-in model to a context-out one. The main advantage of this approach is that, never mind the model is, it can be placed context-out. Our implementation of a Domain Analyzer is based on the Builder Design Pattern [7] used in order to "separate the construction of a complex object from its representation so that the same construction process can create different representations". In our implementation, we use the XML language in order to define context-out models [13, 5].

Inheritance Hierarchy Management

In order to avoid a properties repetition inside a same kind of models, a models library must deal with an inheritance between the stored models. This inheritance between a parent model and its children allows to store these shared properties inside some special models (parent models), dramatically simplifies the children models, and facilitates the maintenance of the whole library. Furthermore, this inheritance hierarchy inside a library provides all the classical benefits of object inheritance: automatic properties transmission, methods overloading if components are described using algorithmic functions. For R.C Rosenberg, "the importance of a good models library is that the model designer can be supplied with reasonable alternatives" [11]. The choice between these alternatives can be strongly facilitated if the inheritance hierarchy is structured in a smart way. In our architecture, the management of inheritance between models is performed using the characteristics of XML. For instance, a ModelFile will inherit a part of its description from an IIM only using a tag substitution or adding.

THE DESIGN AND TEST LIBRARY

Structure of the Library

In order to take into account all the specificities of this kind of libraries, we introduce two new objects. The first of them is an object called "Bench" and is a specialization of the ModelFile object (in the UML meaning [6]). This object includes a new attribute called "results" containing the results of the bench (Figure 3). We use this class instead of the ModelFile in the "Testbench" branch. The second object we introduce is called the "testbench matrix". This matrix is built in a strictly identical way than the abstraction matrix. A design is associated with its corresponding testbench using integer values. The only difference in this case is that these values can be only boolean values since there is no test hierarchy.

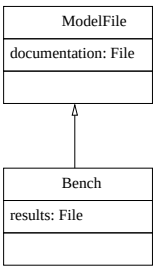

Figure 3: The Bench object.

The first step in order to define our storage architecture using the previously defined notions is to define an element Library. Let's call it "Design & Test". Then, we have to define the domains able to be stored in this library. Let's create the "DEVS", "VHDL" and "Verilog" domains. Using this approach, we are able to store two different formats in a single library. From now, we focus only on the VHDL domain.

The next step is to define application domains linked to the domains. We choose here to define two different application domains: "design" and "testbench". In the first one, we will store the VHDL code describing a design, while in the second one, we will store the corresponding testbenches. We see here a first link between stored elements. We call it a "test link". We saw previously that a description can occur at four different levels. Following this point, we define four CIMs inside the two application domains. We see here a second link between stored models concerning the abstraction hierarchy. We call it an "abstraction link".

Figure 4 presents this architecture. We added two Inheritance Intermediate Models ("RAM" and "ROM") allowing a sharing of some properties between their respective children. What is important in this figure is that we can illustrate the links betweens the elements. "RAM1.1" and "RAM1.2" are linked by abstraction. That means that they represent the same system described at two different abstraction levels (respectively RTL level and Algorithmic level). "RAM1.1" is also test-linked with "RAM1.3". That means that, starting from the key of RAM1.1, we can find immediately its associated testbench.

Format of a Stored Description

In order to illustrate how a HDL description is stored inside a ModelFile element (and also a Bench element), let's take the example of a counter. A very simplified code can be the following:

```
library IEEE;
use IEEE.std_logic_1164.all
```

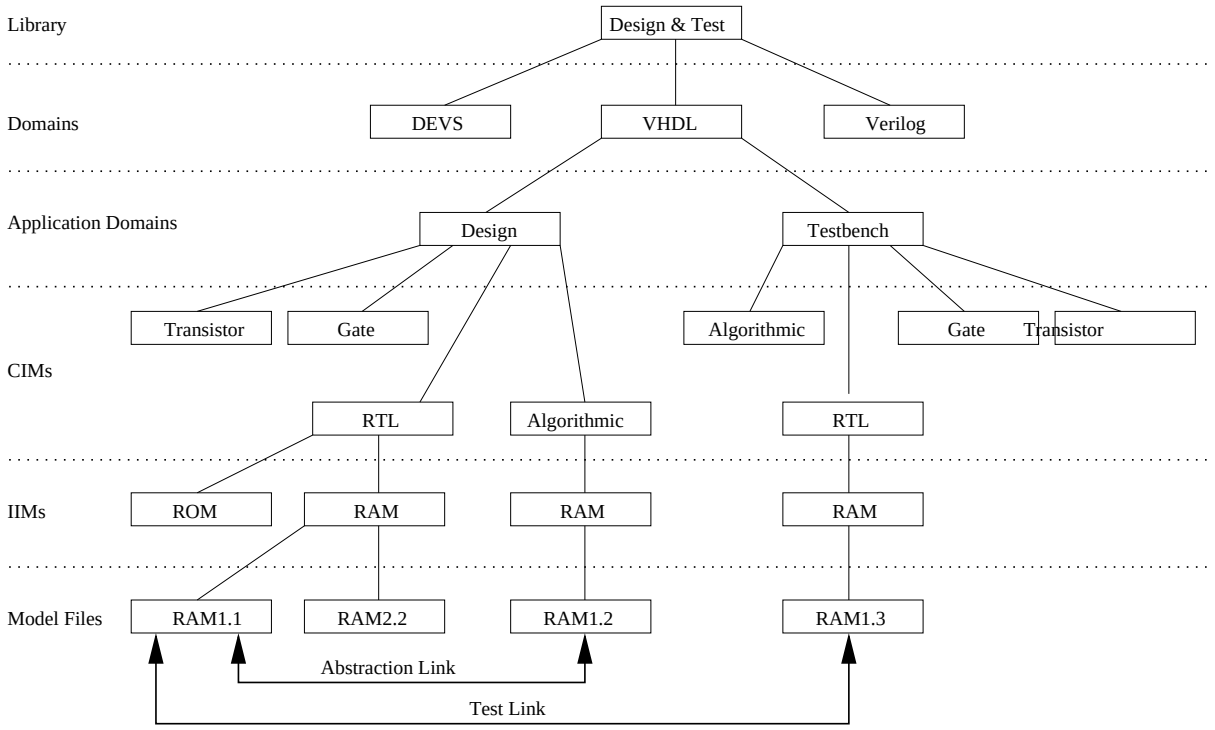

Figure 4: Architecture of the design and test library.

```
ENTITY counter
  PORT(clk, en, clr: IN std_logic;
  rco: OUT std_logic;
  q: OUT INTEGER RANGE 0 TO 9);
END counter;

ARCHITECTURE behav OF counter IS
SIGNAL cnt: INTEGER RANGE 0 TO 9;
BEGIN
  ...
END behav;
```

This description is clearly divided in three parts: the header and the ENTITY and ARCHITECTURE declarations. In this case, the Domain Parser is configured so as to be able to recognize these parts. Once they are identified (following some keywords like ENTITY or constructions like “END *;”), it will transform this context-in description in a context-out one using the XML language as follows:

```
<?xml version="1.0" encoding "UTF-8">
<Description key=12345 Name=counter>
  <header>
    library IEEE;
    use IEEE.std_logic_1164.all
  </header>
  <entity>
    ENTITY counter
    PORT(clk, en, clr: IN std_logic;
    rco: OUT std_logic;
    q: OUT INTEGER RANGE 0 TO 9);
    END counter;
  </entity>
  <architecture>
    ARCHITECTURE behav OF counter IS
    SIGNAL cnt: INTEGER RANGE 0 TO 9;
    BEGIN
      ...
    END behav;
  </architecture>
</Description>
```

Since testbenches are written in VHDL, the same Domain Parser can be used for this kind of descriptions.

This XML description can then be associated with a ModelFile instance and stored in a Library instance.

WEB-BASED IMPLEMENTATION

Kuljis and Paul claim that the web “can serve as an operating system, and as a distribution channel for applications” [8].

They note that the main characteristics of the web are: ease of navigation and use, ease of publishing content, new distribution models and enabling of a nework-centric computing paradigm. Our Libraries Architecture is built on a core storage engine, and provides a set of interfaces enabling a remote access. This is one of the main features of our approach since it allows a design team to work on the same models stored on a storage server. The basic idea is that the storage engine is running on a server and that the client access the models using a Web browser or directly from the environment upon a local network or even over the Internet, following the classical 3-tiers architecture. Clients do not need to know how the models are stored on the server, they should only access them for consulting, adding or removing them. We performed the implementation of these concepts using the Java language and we wanted to provide to the final user the easiest way to access the models libraries remotely. So, we used a servlet/applet approach for the remote part of our work. A servlet [9] is a Java program running on a Web server accept-ing requests from a client, performing some tasks and returning the result, and an applet is a Java program running in the client browser and allowing it to dialog with the server.

CONCLUSION AND PERSPECTIVES OF WORK

This paper presents an efficient approach for reuse oriented Design and Test of digital systems. This work is based upon the concepts of hierarchical object oriented libraries. We presented here a library architecture used for storing, accessing and reusing VHDL descriptions and associated testbenches. The implementation is based on the one hand on the use of object oriented paradigms and on the other hand on the use of XML. Using such an architecture is helpful in two situations. First, the designer can reuse the stored designs directly in their simulation environment using the Domain Parser. Secondly, he can build a new testbench and compare the results with those already stored.

The future orientations of our research work will deal with the following points :

- Integration of the work presented in this paper in a more general framework allowing a web oriented use of a VHDL design and Test library;
- Definition of a software tool allowing testbenches to be automatically derived for an interconnection of basic VHDL components from a set of individual testbenches stored in a library and associated with basic components.

References

[1] P. Ashenden. *The Designer Guide to VHDL*. Morgan Kaufmann Publishers, 2001.

[2] O. Balci, A. Bertelrud, C. Esterbrook, and R. Nance. Developing a Library of Reusable Model Components by Using the Visual Simulation Environment. In *Proceedings of the SSC'97*, 1997. San Diego, CA, USA.

[3] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1992.

[4] P. Benjamin, J. Erratungla, D. Delen, and R. Mayer. Simulation Modeling at Multiple Levels of Abstraction. In *Proceedings of WSC'98*, 1998.

[5] G. Bierman. *Using XML as an Object Interchange Format*. ODMG, Object Data Management Group, 2000.

[6] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2002.

[8] J. Kuljis and R. Paul. A Review of Web-Based Simulation: Wither We Wander ? In *Proceedings of WSC 2000*, 2000.

[9] K. Moss. *Java Servlets, Second Edition*. McGraw-Hill, 1999.

[10] D. Pellerin and D. Taylor. *Vhdl Made Easy !* Prentice-Hall PTR, 1997.

[11] R. Rosenberg. The Bond Graph as an Unified Database for Engineering System Design. *Journal of Engineering for Industry*, 97, 1975.

[12] A. Stritzinger. A Component-Based Modeling Approach. In *Proceedings of WOON'96*, 1996.

[13] W3C Consortium. *Extensible Markup Language (XML) 1.0*, 1998.