Object-Oriented Reuse Methodology for VHDL

Cristina Barna
Forschungszentrum Informatik
Haid-und Neustr.10-14, 76137 Karlsruhe, Germany
barna@fzi.de

Wolfgang Rosenstiel
Universität Tübingen
Sand 13, 72076 Tübingen, Germany
rosen@fzi.de

Abstract

In the reuse domain, the necessity of finding a new, more suitable description language opposes the need to make reuse an accepted practice, and thus related to standards. This paper presents a new method to reuse VHDL described components in an IP centric manner. The basic object reuse model uses an object-oriented extension of VHDL, Objective VHDL. In contrast to conventional reuse approaches, which imply a considerable re-design effort, this new approach bridges the gap between design and reuse integration. The methodology is implemented in the form of a Reuse Management System which handles the classification, modification, adaption, storage and retrieval of the reuse components.

1 Introduction

Reuse of intellectual property in form of hardware models is an important issue in hardware design. The reuse methodologies have evolved in strict interconnection with the definition of new means of specification, expressed by hardware description modalities that permitted a more abstract view on the reusable parts. Taking a look at the historical development of the hardware description methodologies, it is obvious that the trend is to reach a higher level of the description. There were, depending on the level of abstraction reached, various languages employed for describing and simulating electronic designs. The attention was focused on SSI-MSI boards, afterwards on MOS ICs, block-switch and RTL gate-switches, then market driven complex technologies evolved, implying more complex modalities of design. At each and every level, reusing designs was a central issue and a matter of course in the next evolution step.

Present day research addresses the ways of specifying future systems in order to ease automation in design, in particular languages for describing hardware components and methodologies to store and retrieve objects in a database of reusable designs [9]. The complexity of the components available for reuse now requires mechanisms for adaptation; furthermore, communication between components becomes an issue [8]. There are several attempts to find the best language and methodology for description, e.g. existing proposals are SLDL, SpecC, Java. Most of the researchers also propose complete systems for integrating the IP components in designs and focus on verification and validation at a very high level of abstraction[6].

Our goal is to make possible the reuse of firm components, that is, already available designs, which we do not need to redescribe in a new language. That would speed up the creation of a public available database, as it would only require an effort of creating the adaptation modules. The present-day standard, which in Europe takes about 90% of the market, is VHDL. We concentrate on reusing VHDL described components, basing on an object-oriented variant of the language, Objective VHDL.

This work has been partially funded by the BMBF as part of the MEDEA project Euripides.

2 Model

The idea of IP centric reuse focuses on integrating components in a functional assembly, as close as possible to the specification. The attempts to perform reuse in the software domain led to the object-orientation and component software paradigms. The successful way of achieving this was to extend existing languages to be object-oriented, as it was the case with C. In the hardware domain, there were several attempts to add object-orientation to the most popular description language, VHDL (see [4], [3], [2], [7]). We rely on the Objective VHDL description language, shortly OVHDL, in order to define and manipulate the reusable objects. The kernel of the reusable IP will still be native VHDL, in the form one can expect to get from an IP provider, while the wrapper will be written in Objective VHDL (see Figure 1).

In this manner, reuse is made possible for existing descriptions, while still taking advantage of the object-oriented features, like inheritance, polymorphism and overloading, that are supported by this extension of VHDL. We will present the object-oriented elements we use in defining the containers. We will only give a few elements of the

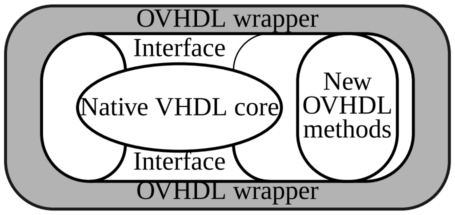


Fig. 1 Basic reuse component

Objective VHDL syntax, as it is required for the sustaining of our model. For further details on the Objective VHDL language definition, please refer to [7].

3 Definition elements and language extensions

The extension to VHDL called Objective VHDL introduces the class concept, class inheritance, type polymorphism and method call or message passing into VHDL, while keeping the VHDL concurrence. We will consequently present some object-oriented constructs, which are new to VHDL and we are using in our model. We will be using the extended backus naur form to introduce the syntax extensions.

An inheritance mechanisms for entities is provided, in the form of the *derived entities* (see definition in Listing 1). A given derived entity declaration may be shared by multiple design entities, each of which has a different architecture. The inheritance is non-cyclic. A derived entity is a new kind of primary unit, similar to simple entities.

```
derived_entity::=
    ‘entity‘ identifier ‘is‘‘new‘ name ‘with‘
        header
        declarative_part
        [‘begin‘
            statement_part]
    end [‘entity‘] [name]‘;‘
```

Lst. 1 Derived entity

A derived entity *header* shares the syntax of an entity header, having the potential to declare ports and generics for communication with the environment. The port and generic lists of the derived entity header will be joined to the lists of its ancestor to form the effective ports and generics lists of the derived entity.

The derived entity *declarative part* allows to declare items that are common to all designs that have the derived entity as their interface. The effective identity statement part is the union of all concurrent statements from the derived entity statement part, all non-labelled concurrent statements from the effective entity statement part of its ancestor entity and all labelled concurrent statements from the effective entity statement part of the ancestor entity whose labels do not occur as a label of a concurrent statement of the derived entity statement part.

The VHDL architecture body (Listing 2) is extended by an optional additional keyword *abstract*. An abstract architecture together with its corresponding entity may contain subprogram declarations without a corresponding subprogram body.

```
architecture_body::=
    ‘architecture‘ identifier ‘of‘ name ‘is‘[‘abstract‘]
        declarative_part
    ‘begin‘
        statement_part
    ‘end‘ [‘architecture‘] [name]
```

Lst. 2 Architecture body

The derived architecture body must report to an entity (derived on not) and an architecture. The ancestor architecture must report to an ancestor entity, or to the same entity as the derived architecture. An abstract architecture cannot be used to form and instantiate an entity, however, a derived architecture body can supply the missing definitions for the declared subprograms.

The VHDL type system has been extended by adding the *class type*. It represents a collection of class attributes (data fields) together with the associated functionality provided by subprograms (methods).

```
class_type::=
    [‘abstract‘]‘class‘
        class_attributes
        class_type_declarative_part
        class_type_object_configuration
    ‘end‘‘class‘[name]
```

Lst. 3 Class type

A value of a class type can be assigned to a signal as well as to a variable of that class type. Polymorphism enables uniform handling of objects related by inheritance. In particular, existing source code can handle additional derived classes without modification. A class type consists of common declarative items and specific declarations targeting an instantiation of the class type as signal, variable or constant. Declarations of data fields are preceded by the keywords *class attribute*, as opposite to signals, variables and constants, which are themselves objects obtained by initializing a class. The class body primarily comprises subprogram body definitions of the subprograms declared in the class type declaration. Moreover, it allows to declare private items. The class configuration implementation distinguishes the subprogram bodies working on a class type instantiation as signal from a class type instantiation as variable or constant.

Class-wide types are deduced by application of the attribute ‘CLASS. T’CLASS denotes the set of types including T and all descendant of T. In case of a class-wide type, the subprogram to be executed is selected from the class which is denoted by the actual type tag of the class.

4 Extensions to the RMS environment

The *Reuse Management System* RMS is a storage, adaption and retrieval environment for manipulating the reuse objects [11]. We extended it in order to handle the firm IP cores, together with the class-wide containers. It bases on an object-oriented model which I will summarily present in the following.

In the RMS terminology, a reusable IP is called a *Component Environment* (CE), composed of a *Component Core* - which is our future commercial firm IP core, and an *Environment*, which will be supplied by the RMS and contains the binding of the IP core to an abstract Objective class.

A *Characteristic Attribute* (CA) $\tilde{C}$ represents a feature describing a reuse component (like *add*, *multiply*). The assignment of a CE to a CA is provided with a weight $g \in I\!N$, which indicates how good the CE fulfils the CA's feature.

$$f_{\tilde{C}}(k) = \begin{cases} g(1 \le g \le L) & \text{if } k \text{ fulfils } \tilde{C} \\ & \text{by intensity } g \\ 0 & \text{otherwise} \end{cases}$$

A subset of CAs can be added into a *Vector of Characteristic Attributes* (VCA) $C = (\tilde{C}_1, \ldots, \tilde{C}_n)$. The assignment of a CA to a VCA is also provided with a weight $|\tilde{C}_i| \in I\!N$. Based on these elements, the RMS creates a classification graph. The reusable components are the leaves of this classification graph.

The similarity between CEs is defined as a function of the weights on the path between the CEs in the classification graph. In the state of the art version of the RMS, the classification graph has a depth of three, so that every two connected (thus similar) components are separated by four branches. The original way of defining it was by simply adding the four weights, which eventually led to the erroneous result that a component was more similar to another than to itself (because the similarity between the two components was computed as the sum of four weights, one of which was the weight g). A second unpleasant effect was that the maximum similarity L was exceeded during this calculus. We introduce a new pondered function to define the similarity of two CEs, expressed by:

$$g_f(k_1, k_2) = \text{average}|g_{C_i}(k_1, k_2)| \quad (1 \le i \le m),$$

where

$$g_C(k_1, k_2) = f_{\tilde{C}_1}(k_1)|\tilde{C}_1| + f_{\tilde{C}_2}(k_2)|\tilde{C}_2|$$

is the similarity of the two CEs with respect to the VCA C, and

$$|\tilde{C}_i| = \frac{|\tilde{C}_i|}{L}$$

is the percental weight of the relation between the CA $\tilde{C}_i$ and the VCA C.

All VCAs constitute a *Characteristic Vector* CV. The characteristic attributes and the corresponding vectors within a CV describe the components concerning one aspect, for example *Function*. Within a CV, each component is assigned to exactly one characteristic attribute.

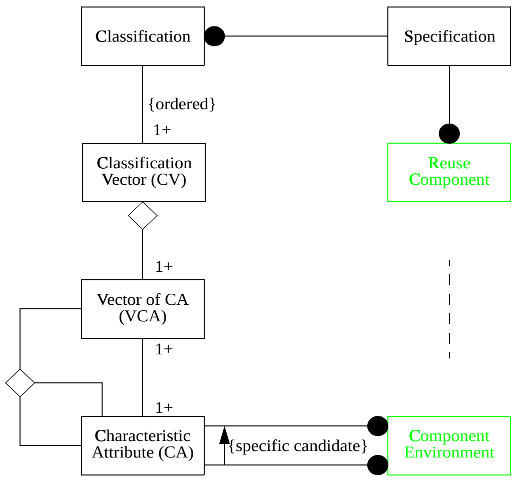


Fig. 2 Classification model

In the initial RMS similarity metric, the similarity was defined only between the reuse components CE. This was extended by the *conceptual similarity*, which defines the similarity between two characteristic attributes related to the same vector. Due to the is-a semantic between a taxonomy node and his parent, each node can be considered as the generalization of its child nodes, and this makes the RMS taxonomy a specialization hierarchy.

We further enlarged the RMS by defining conceptual similarities between the internal nodes of the library graph. The IPs attached to these nodes correspond to complex or abstract components, and following, the derived child nodes will be in a is-a or has-parts relation to them.

The system searches for matching and similar IPs, the selected node and all child nodes are evaluated according to the similarity metric. For the matching and similar IPs the attributes are now checked. For similar IPs, only the relevant attributes are checked.

5 The two layer model

The model we use is centred on a firm IP core which will be stored in a library of reusable objects (see Figure 1). This kind of reuse is called grey-box reuse. It allows controlled modifications, but they do not affect the reused code directly. The grey-box reuse is strongly correlated to the object-oriented paradigm.

The components will be included into classes that provide the necessary level of abstraction. A class, in its object-oriented meaning, provides an abstract frame for the integration of the IP cores. Objects in the definition of the LRM [1] are signals, variables and ports. We define objects as entity-relationship pairs.

```
entity comparator is
     generic (Width: Positive);
          port (   a,b: in bit_vector(Width-1 downto 0);
                   a_gt_b: out bit);
end comparator;
```

Lst. 4 Class container

We accept the classes to be object containers, described in Objective VHDL (see Listing 1). They reflect the functional cover of a singular kind of components (e.g. comparators). With respect to the structure of our RMS, they are designated by CAs. The firm IP cores are characterized by means of their communication with the environment. This information needs to be provided by the designer (owner) of the IP.

Following, the cores that have a similar functionality will be subordinated to this class container, as the components that will contain them are derived by inheritance and extension from the container, using the mechanisms described in the previous paragraphs.

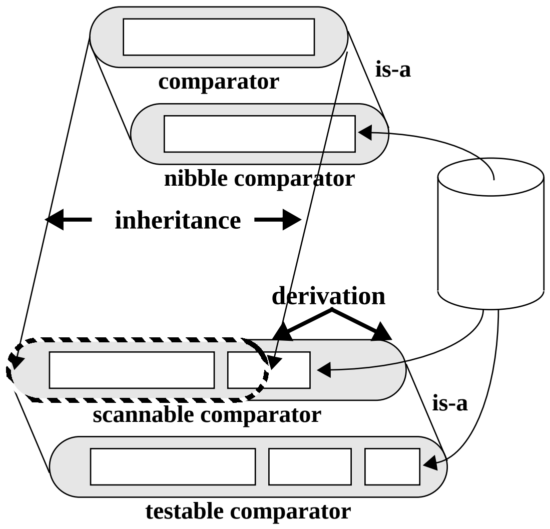


Fig. 3 Object composition

We store two kinds of objects in the database:

- there are class containers, which are abstract components; they only have a formal semantic corresponding to the functional description of the real components, and
- the firm core components $\tilde{C}$ have associated the wrapper $\tilde{W}$, which they inherited from the class container. The database objects are modelled as a tuple $\tilde{N} = \{\tilde{C}, \tilde{W}\}$.

The fitting of a firm core $\tilde{C}$ into $\tilde{W}$ is handled by the interface description $\tilde{D}$. This is a description of the parameters of the IP core $\tilde{C}$, which will further be called *signature.*

The signature describes the type and mode of the parameters, and thus implicitly defines the sequence for the values to be passed as parameters. A set of values will be passed as parameters when calling a method. This behaviour approximation works well in the software domain. Unfortunately, in the hardware domain there are certain restrictions for the application of the signature based modification, that were analysed in [10].

A signature does not describe the fact that an interpretation of a signal's value may change over time; neither does it consider the case that not a single value, but a waveform is given back to a caller. The solution is to separate protocol specification and implementation from the functionality of a method. Re-definition of inherited methods must be done without affecting the protocol parts. Protocol waveforms passed as parameters of mode must not schedule transactions during the execution or after the execution of the method.

The database objects can be modelled as a n-uple $\tilde{C} = \{C_{core}, C_{signature}\}$. We define C_{core} as the VHDL source for the IP C and $C_{signature}$ as the result of extracting the interface information out of the IP C. The definition of signatures respects the interface part of the Objective VHDL description.

```
signature::=
     '('parameter')'semantic';'
parameter::=
     (identifier';'type';'direction)*
```

Lst. 5 Signature definition

Both are keys for the search in our database system, as they are defined as characteristic vectors. Accordingly, the similarity is defined by two factors. First, the classification of the component only places it under the corresponding node of the library graph RL (represented by a CA in the RMS model). These nodes are further connected through weighted paths. The functionality will be stepwise specified by advancing through the classification tree. Additionally to the existing system, we define the possibility of adding more complex components at a higher level of the graph RL, by allowing multiple levels of CA nodes. Secondly, the signatures of the methods are included under a separate classification vector CV.

The process of including a new IP in the database implies analysing the functionality and the interface of the component. We distinguish two cases:

- for the new IP, there is a container describing the same functionality. All we need to do in this case is to create the derived type that respects the signature of all the methods of the new object, by overloading or by simply adding new methods to the container class.

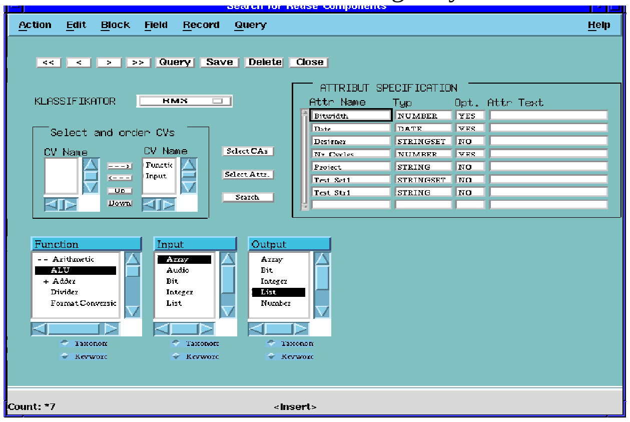
- there is no class that close enough describes the new component, thus the effort of modifying an existing container would be rather equal to that required to create a new class; or, as an other aspect, including the component into an existing functional class is semantically impossible.

In this latter case, the database administrator has to create a new container class, task that can become very complex depending on the interface of the IP. This will be supported by the RMS by providing a template and a clear methodology for integration.

6 Implementation

The model has been verified on common benchmark designs. A special interest was taken in the IEEE1149.1 compliant designs, as future work will address observability and testability issues. The goals of the IEEE Std 1149.1 are to provide a standardized approach to testing the interconnections between integrated circuits, testing the circuit itself and observing circuit activity during the normal operation of the components. We based on the implementation of the basic components IEEE Std 1149.1-1990 in native VHDL [5]. At present, some of these components are described in OVHDL, others, like the TAP controller, have an OVHDL wrapper, and the testbench and the low-level procedures, which constitute the functional library of the TAP, are still native VHDL.

An example implementation addressed interconnecting object-oriented and native VHDL designs. Therefore, a class container for a comparator was defined (see Figure 3) and a nibble-comparator, a scannable and a testable comparator were created by inheriting the wrapper of this class and including the specific boundary scan components. To accommodate the boundary scan register, four ports had to be added to the scannable and the testable comparator, therefore the interface had to be enlarged by derivation.



ATTRIBUT SPECIFICATION

Attr Name	Typ	Opt.	Attr Text
Bitwidth	NUMBER	YES	
Date	DATE	YES	
Designer	STRINGSET	NO	
Nr Cycles	NUMBER	YES	
Project	STRING	NO	
Test Set1	STRINGSET	NO	
Test Str1	STRING	NO	

Fig. 4 Search for reusable components

The Reuse Management System was implemented using the Oracle Server 7.3. The database is running under Solaris 2.51 on a Sparc 20 computer, the application servers are running under NT Server 4.0. A screenshot of our retrieval system is presented in Figure 4. The next version of the RMS will run under Oracle 8.0.

7 Conclusion and future work

The combination of standard and object-oriented VHDL proves to be a successful way of including existing designs into future IP centric systems, respecting the definition of grey-box reuse. While being compliant with the most widely accepted standard description language, the presented model keeps up with object-oriented features like inheritance, polymorphism and overloading, thought as indispensable in future reuse-based design.

Future work will address the automated adaption of the IP cores for design by reuse, as this is still a productivity issue and a matter of return investment planning. Further, the transparent coupling to the high-level synthesis will be considered. In parallel, adding scanability and testability by including IEEE1149.1 compliant modules in an early design phase, will be addressed.

8 References

[1] IEEE Standard VHDL Language Reference Manual. IEEE standard 1076-1993, 1993.

[2] P.J. Ashenden, P.A. Wilsey, and D.E. Martin. SUAVE: Painless Extension for an Object-Oriented VHDL in Rapid Systems Prototyping with VHDL. In *Proc. of the VIUF Fall 1997 Conference*, 1997.

[3] J. Benzakki and B. Djafri. Object-Oriented Extensions to VHDL - the LaMI Proposal. In *Proc. of CHDL'97, Toledo, Spain*, 1997.

[4] D. Cabanis and S. Medhat. Object-Oriented Extensions to VHDL: The Classification Orientation. In *Proc. of the VHDL User Forum Europe 1996*. Shaker Verlag, 1996.

[5] P. Campbell, M. Vai, and Z. Navabi. Implementation of IEEE Std 1149.1-1990 in VHDL. *Proc. of the VHDL International Users' Forum Conference*, 1992.

[6] D. Eisenbiegler and C. Blumenröhr. Gropius - Advanced Reuse Concepts in a New Hardware Description Language. *Proc. of the 2nd GI/ITG/GMM Workshop Reuse Techniques for VLSI Design*, 1998.

[7] S. Maginot, W. Nebel, W. Putzke-Röming, and M. Radetzki. Final Objective VHDL Language Definition. Technical report, LEDA and OFFI, May 1997.

[8] W. Putzke-Röming, M. Radetzki, and W. Nebel. A Flexible Message Passing Mechanism for Objective VHDL. In *Proc. of DATE'98, Paris, France*, Feb. 1998.

[9] A. Reutter, B. Mößner, I. Kreutzer, and W. Rosenstiel. Wiederverwendung komplexer Komponenten für Synthese und Simulation unter Verwendung von VHDL. In L. Peters and K. Lagemann, editors, *Entwurf Integrierter Schaltungen, 8. E.I.S.-Workshop, Tagungsband*, Apr. 1997.

[10] G. Schumacher and W. Nebel. Object-Oriented Modelling of Parallel Hardware Systems. In *Proc. of DATE'98, Paris*, Feb. 1998.

[11] R. Seepold. *A Hardware Design Methodology with Special Emphasis on Reuse and Synthesis*. PhD thesis, Univ. of Tübingen, 1997.