OBJECTIVE VHDL: TOOLS AND APPLICATIONS

Nikhil Singla[1], Nipun Jain[2], Paras Thakral[3]
B. Tech (6th sem), CSE, DCE, Gurgaon

Abstract: Objective VHDL, an object-oriented extension to VHDL, has been designed within the OMI-ESPRIT project REQUEST to facilitate hardware modeling at a higher level of abstraction and with increased potential for reuse. In this paper, we present a pre-processor tool set for the translation of Objective VHDL into VHDL, allowing to use the new language with existing simulation and synthesis environments. Since the back end processing (simulation, synthesis) takes place on VHDL level, the correlation of the generated VHDL to the original Objective VHDL sources should be understood by the user and will therefore be outlined. The presentation is based on the example of object-oriented buffer (FIFO, LIFO) data types that can be synthesized into gates.

I. INTRODUCTION

VHDL stands for VHSIC Hardware Description Language. VHSIC stands for Very High Speed Integrated Circuit. It was a DoD program aiming at a new mean to describe electronic systems without any dependency on implementation. One of the initial purpose (just like the typical government) is to have an unified way of documenting hardware designs. Just like any other government programs, a lot of money has been invested in the development. It is estimated 17 millions was devoted to the direct development of VHDL and another 16 millions for VHDL design tools. It started in 1981. The major funding was given to a team consists of IBM, Intermetrics, and TI to develop VHDL in 1983. In August 1985, a final document was release which defines VHDL version 7.2. By 1987, it has been adopted as IEEE standard - IEEE 1076. It was again modified in 1993 and is named IEEE std 1076-1993. This modification is upwardly compatible. It removes some ambiguities in the language and added some new capabilities. Use of object-oriented techniques is being discussed and applied [1][12] as a means to increase the degree of abstraction and reusability of hardware models. To support this goal, a couple of proposals to extend the hardware description language VHDL [7] for object-oriented constructs have been made since the early nineties [2][3][4][5][11][13][14]. Among the more recent ones is Objective VHDL [8][9], for which a preprocessing tool set for translation into VHDL is currently being developed so as to make the new language applicable in practice using standard VHDL simulation and synthesis. Whereas the preprocessing approach allows us to use a large set of VHDL back end tools and to link Objective VHDL into VHDL based design flows, it has its shortcomings as well. Thus, the aim of this paper is to not only present the Objective VHDL tool set and an example of its application, but also the correlations between original sources and the results of translation in order to help potential users to under-stand the translation.

II. OBJECTIVE VHDL TOOL ARCHITECTURE

The translation from Objective VHDL into VHDL is implemented based on an extension of the LEDA VHDL System (LVS) for Objective VHDL, see Fig. 1. The LVS analyzer parses Objective VHDL source code, analyzes it semantically, and stores the resulting attributed abstract syntax tree in an intermediate format (VIF) accessible via a procedural interface (LPI). The translator, when invoked, transforms the parts of the intermediate format which correspond to Objective VHDL constructs into a standard VHDL representation. Finally, a reverse analyzer generates from this intermediate for-mat VHDL source code that can be imported into other VHDL environments. This architecture also provides the opportunity to apply VHDL back end tools that work directly with the generated VIF, or to even implement native Objective VHDL back ends that operate with the original VIF generated by the analyzer.

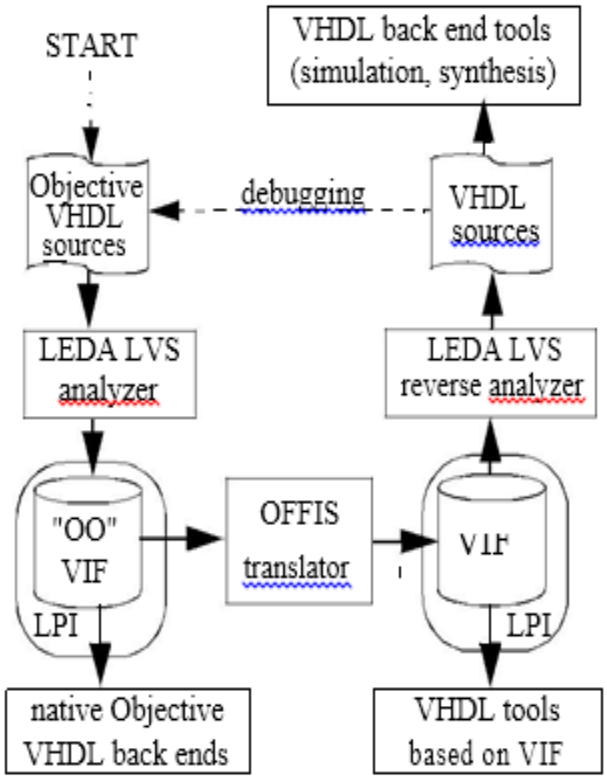

Fig. 1: Objective VHDL tool architecture

III. CLASS TYPES AND THEIR TRANSLATION

An Objective VHDL class type can contain class attribute declarations representing data fields of the individual objects, subprogram declarations and bodies representing the methods of the class, and certain other declarations known from VHDL, namely type, subtype, constant, use clause, and alias declarations. In the following, we show with the buffer example the use of these declarations to model functionality, and their translation into plain standard VHDL as performed by the Objective VHDL to VHDL translator.

A. Regular VHDL declarations

Besides subprograms (methods) and class attributes, Objective VHDL class type or derived class type declarations can contain type, subtype, constant, use clause, and alias declarations to be used in the class. In the buffer example, we declare an array type, BUFFER_ARRAY, which will be used to store the buffer contents (see Listing 1). The size of the array is currently declared as a constant outside the class type; we plan, however, to allow the declaration of generic constants (like entity generics) in class types in order to provide improved parameterization.

Listing 1: Abstract class type BASE_BUFFER
package BUFFER_PKG is constant SIZE : POSITIVE := 8;
type BASE_BUFFER is abstract class
type BUFFER_ARRAY is array (0 to SIZE - 1) of INTEGER; -- continued in Listing 3

In the example, the class type BASE_BUFFER is declared as abstract. This means that it can only be used to derive other types from, but no instance of it can ever be created. In consequence, we do not need to provide a translation of the abstract class type on VHDL level. Only the declarations contained in the class must be translated because they might be used in the derived classes.

Regular VHDL declarations such as the array type declaration are translated by moving them into the declarative region in which the class declaration is contained. In our example, the array type will thus be moved into the package BUFFER_PKG (see Listing 2). If necessary, the moved declaration would be renamed to avoid conflicts with declarations having the same name. Then, of course, also wherever the renamed declaration is used, an adjustment to use the new name must be performed.

Listing 2: Translation of BASE_BUFFER (translated statements in bold letters) package BUFFER_PKG is constant SIZE : POSITIVE := 8; type BUFFER_ARRAY is array (0 to SIZE - 1) of INTEGER; end BUFFER_PKG;

B. Class attribute declarations

Consider class BASE_BUFFER and classes LIFO and FIFO derived from it. BASE_BUFFER declares the class attribute STORAGE using type BUFFER_ARRAY mentioned before. This class at-tribute is inherited by FIFO and LIFO and will be used as the storage of the buffer. LIFO uses STORAGE like a stack and therefore declares the class attribute INDEX which will point to the top element of the stack. FIFO uses STORAGE like a ring buffer, declaring class attributes FIRST and LAST which will point to the first and last element in the ring buffer, respectively:

Listing 3: Abstract class type BASE_BUFFER (cont'd from Listing 1) class attribute STORAGE : BUFFER_ARRAY := (others => 0); -- continued in Listing 6

Listing 4: Derived class types LIFO and FIFO package LIFO_PKG is type LIFO is new class BASE_BUFFER with class attribute INDEX: NATURAL range 0 to SIZE := 0; end class LIFO; end package LIFO_PKG; package FIFO_PKG is type FIFO is new class BASE_BUFFER with class attribute FIRST : NATURAL range 0 to SIZE - 1 := 0; class attribute LAST : NATURAL range 0 to SIZE - 1 := 0; end class FIFO; end package FIFO_PKG;

The class attribute declarations of a class type are transformed into record elements and collected in a record declaration. In case of a derived class, this record does not only contain elements correspond-ing to the class attributes declared in the derived class itself, but also elements representing the class attributes inherited from the parent class (and, transitively, the parents thereof, if any). Thus, the record generated from class FIFO has elements STORAGE, FIRST, and LAST (see Listing 5) The record generated from LIFO, which is not shown here, has elements STORAGE and INDEX.

Since VHDL record elements cannot have an initialization expression, initialization expressions of the class attributes are collected in an aggregate constant which will be used by the translator to initialize instances of the record. In case of the FIFO, constant INIT_CONST_FIFO of type FIFO is generated for that purpose.

Listing 5: Translation of FIFO package FIFO_PKG is type FIFO is record STORAGE : BUFFER_ARRAY; FIRST : NATURAL range 0 to (SIZE - 1); LAST : NATURAL range 0 to (SIZE - 1); end record ; constant INIT_CONST_FIFO : FIFO := ((others => 0), 0, 0);
-- continued in Listing 7

C. Method (subprogram) declarations

Subprograms can be declared in class types using normal VHDL syntax. These subprograms are interpreted as methods of the class. A buffer may have methods IS_FULL, IS_EMPTY, PUT, and GET as declared in class BASE_BUFFER (see Listing 6). Whereas IS_FULL and IS_EMPTY are declared for any instantiation of the class type (i.e. as a signal, a variable, or a constant), PUT and GET are declared for variable only. This means that they can only be invoked with an object that is a variable, and that they need to be implemented only for use with variables. Note that these method subprogram declarations are inherited by the derived classes LIFO and FIFO.

IV. POLYMORPHISM

The concept of polymorphism is provided by Objective VHDL through the attribute 'CLASS that re-turns the so-called class-wide type. If T is a class type (derived or not, abstract or not), T'CLASS is a type compatible not only with T, but also with all class types derived from T. If a signal, variable, or constant is declared with this class-wide type, any value of type T or derived can be assigned to it. Furthermore, methods declared in T can be invoked with an instance of the class-wide type. For in-stance (cf. Listing 13), assuming variable B is of type BASE_BUFFER'CLASS, method PUT can be invoked with B. Then, depending on the value stored in B, which may either be of type LIFO or FIFO, the respective implementation of PUT corresponding to LIFO or FIFO is executed. This mechanism is known as dynamic binding. We will devise translation schemes for the class-wide type, assignment, and dynamic binding subsequently.

A. Class-wide type

When a class-wide type T'CLASS is used in an Objective VHDL model, the translator will generate in the WORK library a new package named T_POLYM_PKG to contain all the declarations resulting from the translation of T'CLASS (see Listing 10). If this name already exists in the WORK library, a suffix will be appended to create a unique name. Remarkably, there is some overhead in this data representation because the mutual exclusion of FIFO and LIFO is not exploited. A value stored in an instance of BASE_BUFFER_CLASS may either be a FIFO or a LIFO, but not both at a time. Thus, elements FIRST and LAST could be stored using the same resources as for INDEX. This, however, is not possible using high-level data types such as INTEGER or REAL due to VHDL's strong typing. We therefore plan to implement an additional mapping of class attributes onto a bit-level representation that will allow sharing resources for efficient synthesis, but on the other hand impair comprehensibility of the generated VHDL code.

V. USE OF OBJECT-ORIENTED DATA TYPES

Object-oriented programming embodies the three principles of encapsulation, polymorphism and inheritance. Let's define these three principles in more detail. Encapsulation is the mechanism used for information hiding. Encpasulation allows access to the contents of a data structure only through function calls which operate on that data. In VHDL, we can group a data structure and its associated functions in a package. However, a VHDL package is an abstract data type (ADT) when used in this way. VHDL does not support encapsulation through a package because the data structure is accessible without recourse to the package's functions. For example,

```
package unencapsulated_ADT is
 subtype index is integer range 0 to 31;
 type colour is (RED, YELLOW, GREEN, BLUE);
 type access_mode is (READ, WRITE);

 type super_colour is record
  index_element : index;
  colour_element : colour;
 end record;

 procedure set_colour_element (
  super_colour_parameter: inout super_colour;
  colour_parameter: colour
 );

 function get_colour_element (
  super_colour_parameter: super_colour
 ) return colour;
end unencapsulated_ADT;

package body unencapsulated_ADT is

 procedure set_colour_element (
  super_colour_parameter: inout super_colour;
  colour_parameter: colour
 ) is
 begin
  super_colour_parameter.colour_element :=
colour_parameter;
 end set_colour_element;

 function get_colour_element (
  super_colour_parameter: super_colour
 ) return colour is
 begin
  return super_colour_parameter.colour_element;
 end get_colour_element;

end unencapsulated_ADT;
```

In use, one would expect to see a process statement something like,

```
 a_label: process (a_super_colour)
  variable a_colour: colour;
 begin
  a_colour := get_colour_element(a_super_colour);
 end process;
```

However, this relies on programmer discipline. The package does allow direct access to the variable's data structure - this is not encapsulation. The following code snippett is an example of direct access,

```
 b_label: process (sensitivity_list)
  use a_library.unencapsulated_ADT.all;
  variable a_colour: super_colour;
 begin
  a_colour.colour_element := RED;
 end process;
```

To extend the abstract data type (ADT) principle to encapsulation, the data structure declaration and its associated access functions need to be defined in a new kind of programming element, typically referred to as a class. The term class will be used throughout this article to denote a progamming element that contains both a data structure declaration and its associated functions. In VHDL, a package serves as such a programming element, notwithstanding its information hiding limitation. In object-oriented

programming, an object is an instance of a class. We can draw a parallel between OOP and VHDL already in that a variable (or signal) is an instance of a type. If the declaration of this type occurs in a package along with functions applicable to that type, then in order to use VHDL variables as objects in an OOP sense, we need only make the contents of the package visible at the same level of scope as the variable.

VI. CONCLUSION

We have presented the application of Objective VHDL to hardware modeling using object-oriented data types to describe different kinds of buffers. Objective VHDL tools, of which a free limited demo version is now available, allow to translate the object-oriented models into plain standard VHDL. Pro-vided that the object-oriented models suitably use the object-oriented constructs together with modeling styles known from VHDL, it is possible to have a gate netlist synthesized from the translation result. As we have demonstrated, there may yet be some overhead incurred by the transformations applied to polymorphism. However, optimization techniques have been developed and will be implemented to further optimize the result of translation for synthesis. Further, in a future project, we plan to formalize the notion of synthesizability for Objective VHDL. Since the focus within this paper has been on the object-oriented data types of Objective VHDL and to show that, after application of the preprocessing tool set, they can be synthesized into hardware, we wish to emphasize two points that have been neglected here. First, Objective VHDL does also pro-vide object-orientation with entities and architectures by adding inheritance to VHDL's structural modeling capabilities. Second, there are applications such as writing test benches or high-level simulation models not intended for synthesis that may require modeling styles and optimizations other than those presented or mentioned here.

REFERENCES

[1] Allara, M. Bombana, P. Cavalloro, W. Nebel, W. Putzke-Röming, M. Radetzki. ATM cell modelling using Objec-tive VHDL. Proc. Asia South Pacific Design Automation Conference (ASP-DAC), 1998, pp. 261-264.

[2] P. J. Ashenden, P. A. Wilsey, D. E. Martin. SUAVE: Painless Extension for an Object-Oriented VHDL. Proc. VHDL Int'l Users' Forum (VIUF, Fall Conference), 1997, pp. 60-67.

[3] J. Benzakki, B. Djafri. Object Oriented Extensions to VHDL—The LaMI proposal. CHDL'97, Toledo, Spain, 1997, 334-347.

[4] D. Cabanis, S. Medhat, N. Weavers. Classification-Orientation for VHDL: A Specification. VHDL International Users' Forum (Spring Conference), Santa Clara, USA, 1996, pp. 265-274.

[5] W. Ecker. An Object-Oriented View of Structural VHDL Description. VHDL International Users' Forum (Spring Conference), Santa Clara, USA, 1996, pp. 255-264.

[6] M. Fowler, K. Scott. UML distilled: applying the standard object modeling language. Addison-Wesley, 1998.

[7] IEEE Standard VHDL Language Reference Manual. IEEE Std 1076-1993.

[8] S. Maginot, W. Nebel, W. Putzke-Röming, M. Radetzki. Final Objective VHDL language definition. REQUEST Deliverable 2.1.A (public), 1997. Available on the WWW from URL http://eis.informatik.uni-oldenburg.de/research/ request.html

[9] M. Radetzki, W. Putzke-Röming, W. Nebel, S. Maginot, J.-M. Bergé, A.-M. Tagant. VHDL language extensions to support abstraction and re-use. Proc. 2nd Workshop on Libraries, Component Modelling, and Quality Assurance. Toledo, Spain, 1997, pp. 47-62.

[10] M. Radetzki, W. Putzke-Röming, W.Nebel. Übersetzung von Objektorientiertem VHDL nach Standard VHDL. Proc. GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltun-gen und Systemen, Paderborn, 1998.

[11] G. Schumacher, W. Nebel. Inheritance Concept for Signals in Object-Oriented Extensions to VHDL. Proceedings of Euro-DAC'95 with Euro-VHDL'95, IEEE Computer Society Press, 1995, pp. 428-435.

[12] G. Schumacher, W. Nebel, W. Putzke, M. Wilmes. Applying Object-Oriented Techniques to Hardware Modelling— A Case Study. Proc. VHDL User Forum Europe, 1996.

[13] S. Swamy, A. Molin, B. Covnot. OO-VHDL. Object-Oriented Extensions to VHDL. IEEE Computer, October 1995, 18-26.

[14] R. Zippelius, K. D. Müller-Glaser. An Object-oriented Extension of VHDL. VHDL Forum for CAD in Europe (Spring Conference), 1992, pp. 155-163.

