HANDLING COMPLEX VHDL SEMANTICS WITH AN OO INTERMEDIATE FORMAT

*Dara Rahmati, Abolfazl Salimi Zebardast, Mohammad H. Reshadi, * Zainalabedin Navabi*
Electrical and Computer Engineering Department, Faculty of Engineering / University of Tehran / Tehran, Iran
{dara, salimi, reshadi}@cad.ece.ut.ac.ir
* Northeastern University / Boston, MA 02115
Tel: 617-373-3034; Fax: 617-373-8970; navabi@ece.neu.edu

ABSTRACT

One of the most important problems for integrating CAD tools is sharing the design information between various tools and environments. Using a standard intermediate format makes the interfacing and processing very easy so that all the applications may start their own processing from Intermediate formats. We have used CHIRE (Compiled HDL Intermediate Representation with Extensibility) intermediate format, which is a revision of AIRE/CE[1] and have implemented a VHDL analyzer that generates the CHIRE intermediate format and covers the whole grammar. It also has a powerful semantic checking capability. In this paper we will propose our algorithm of semantic checking and illustrate its steps using a very complicated example for challenging aspects of semantic check in VHDL. Being a good example of using CHIRE, It also implicitly demonstrates the benefits of using an object-oriented format for CAD tools.

1. INTRODUCTION

There are many unique features in VHDL that makes its processing very difficult. It has a context sensitive grammar and is a strongly typed language. On the other hand function overloading in VHDL can be done even through return type or actual parameter indications. Array and record aggregates, different kinds of parameter association, slice names, resolution function and universal types are other complex topics in a VHDL analyzer. Using an OOIF (Object Oriented intermediate format) such as CHIRE has two important aspects, first; through object oriented techniques, complex tasks mentioned, are made easy and manageable; and second, analyzer can focus on covering the whole grammar and producing a CHIRE structure in the memory that can be used by different tools which work based on VHDL. In Section 2 and 3 we propose a brief description of CHIRE OOIF and the essential points of VHDL analyzer and its unique features and the difficulties arises in implementing a VHDL analyzer, since understanding the structure of this memory representation and the task of analyzer is essential in order to understand and use the remaining of the paper. In section 4 we describe our algorithm of semantic checking of VHDL and at last the improvements and conclusion.

2. CHIRE OBJECT ORIENTED INTERMEDIATE FORMAT (OOIF)

Processing a binary format is not only much easier than that of a source code but also provides better means of interoperability and protecting the intellectual property.

CHIRE came to existence when we tried to solve the deficiencies of AIRE/CE (for more information refer to [1][2][3]). This memory representation is an OOIF and is designed to support VHDL, VHDL-AMS, Verilog, and other languages. Worked into CHIRE is extensibility capability, which makes this standard adaptable to new applications.

All classes in CHIRE are inherited directly or indirectly from a class named MR (Memory Representation), and the graph of compiled design (memory representation) has many members of type MR*. There are five levels of inheritance in the hierarchy of CHIRE and they have been designed to achieve the

[1] The version of AIRE/CE (Advanced Intermediate Representation with Extensibility/Common Environment) being discussed in this paper is the version that is publicly circulated on the Web [3], and several companies commercially use versions of AIRE/CE, which may differ from the publicly available draft. AIRE is a trademark of FTL Systems.

best capabilities of object-oriented design concepts such as inheritance, polymorphism and virtual functions and there are about three hundred different classes. (For more information of the detailed structure of the CHIRE go to references [3][6]). In next sections we describe the role of the VHDL analyzer and a perspective of a generated CHIRE memory representation as an example.

```
Entity tester is end tester;
architecture checker of tester is
type bit2 is array(3 to 2*2) of bit;
function f(a,b:bit) return bit2 is
begin
  return ("00");
end;
function f(c,d:bit) return bit2 is
begin
  return ("11");
end;
function "+"(a,b:bit2) return bit2 is
begin
        return ("00");
end;
function "*"(a,b:bit2) return bit2 is
begin
        return ("00");
end;
signal a,b,c,d:bit2;
signal g,h:bit;
begin
  a <= b * ( g, h ) + f( g, d=>h );
end checker;
```

Figure 2-1: An example to demonstrate semantic checking

3. ANALYZER AND SEMANTIC RULES OF VHDL

3.1. Generating CHIRE

Consider the VHDL source code in Figure 2-1. Here the intermediate format that is generated is CHIRE. The summarized memory representation that is generated for the source code is illustrated in Figure 3-1. For every individual concept or construct in VHDL grammar, there usually exists a corresponding CHIRE class. When the analyzer recognizes each syntactic rule, it generates the appropriate instances of the corresponding CHIRE classes in the memory and will set the links between these objects. The process of CHIRE memory representation construction is completed when complex task of semantics checking is done. This is fully described in section 4. So we summarize the tasks of a VHDL analyzer as checking the syntax, producing CHIRE and finally semantic checking.

3.2. VHDL generality and its Context sensitive and complex semantic rules

VHDL is a modeling language, that is, one can model different processes in VHDL, (although it has been mainly designed to be used for modeling digital circuits) and this is possible because of the several utilities it offers to the programmers, such as various user defined types, functions, procedures and virtual functions by some means, Therefore strict restrictions and wide prospect have lead VHDL to one of the most difficult languages to be analyzed (not compiled!). The most important part of semantic checking is type checking. now consider the signal assignment box in Figure 3-1 which is shown in details in Figure 3-2.

The example illustrates the typical semantic check algorithm for all VHDL statements and implicitly shows the benefits of using CHIRE in the syntax tree.

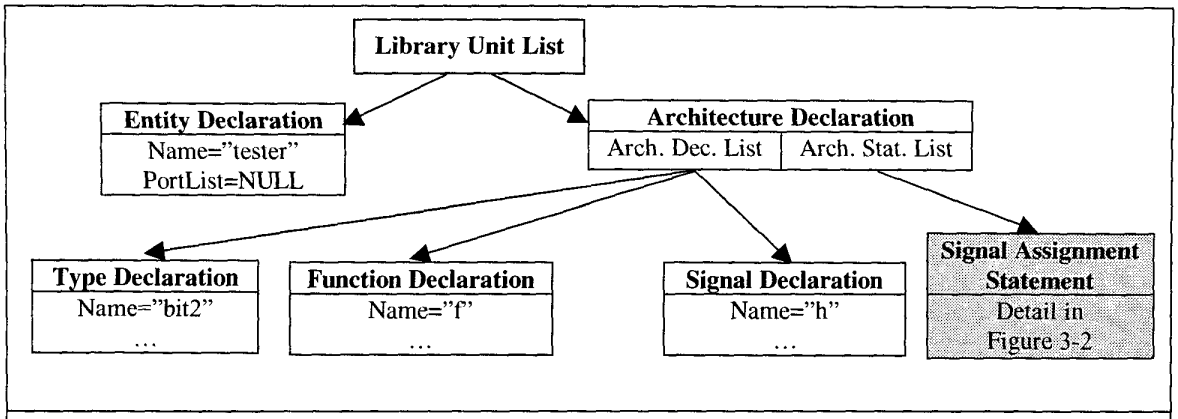

Figure 3-1:The simplified CHIRE memory representation for source code in Figure 2-1, the shaded box will be illustrated in detail in Figure 3-2.

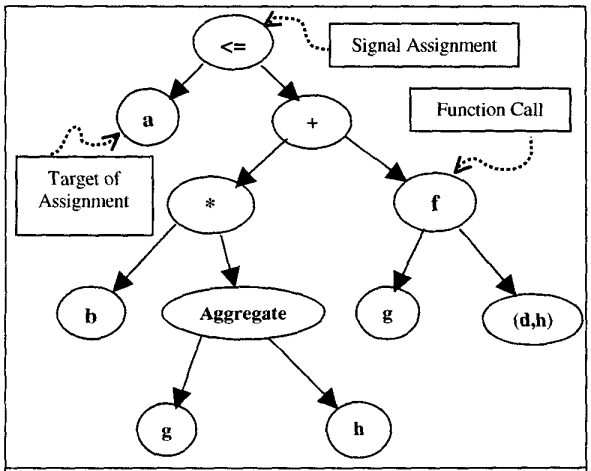

Figure 3-2: The detailed structure for the shaded box in Figure 3-1 and its corresponding signal assignment in Figure 2-1.

4. HANDLING SEMANTIC CHECKING WITH CHIRE

4.1. Statement Semantic Checking

A virtual function named *SemanticCheck()* has been implemented in the super class of all statement classes and is overridden for all kinds of statement classes. This function is called when each of the statements in the analyzer is parsed successfully, i.e. for every kind of statements the corresponding semantic checker is identified and called in runtime to check the semantic restrictions of the statement. In fact we call a single function named *SemanticCheck()* to carry out the whole semantic checking process of a VHDL file. This function will call all the corresponding *SemanticCheck()* functions for the memory representation of other statements. Here we will describe the signal assignment semantic checking as a good example of this process. It also illustrates the other special and powerful concept in our algorithm identified as type checking, which is an essential process to check and manipulate the types of expressions. Figure 4-3 describes the simplified semantic check function for the concurrent conditional signal assignment in VHDL. The *SemanticCheck()* functions, shown in this figure, mainly uses the *TypeCheck()* function. The *TypeCheck()* function is again a virtual function that has been declared virtually in the super class of all expression classes and have been overridden in all expression classes differently to handle the process of type checking. Figure 4-2 shows a template for this function.

4.2. Expression Semantic Checking

Most of the programming languages have a context free grammar and in contrast, VHDL grammar is context sensitive mainly in its expression region, that is, the type of an expression may be defined or changed by each of its sub elements, to address this shortly consider the expression in Figure 4-1.

```
"000"+"001"+"010"+...+"111"+b3
```

Figure 4-1: An example to demonstrate context sensitivity of VHDL grammar

If the type of *b3* is a *bit vector* of length three all other statements are treated as *bit vectors* ,but if *b3* is a *qit vector* (*qit* is a quad type which have four values '0','1','Z','X'), then all other expressions are treated as *qit vectors* and therefore for each of the "+"s the proper operator (predefined or overloaded) should be called, this requires some type lists to be transferred in the process of type checking for expressions. Consider the list transfers in function in Figure 4-2.

```
TypeCheck(PassedTypeList)
{
        ...
        return  ReturnTypeList
}
```

Figure 4-2: A template for describing the TypeCheck() function of different classes

When *TypeCheck()* function is called for an element, The *PassedTypeList* indicates the possible types of the corresponding element. The *ReturnTypeList* includes those types the element can actually be. In this case the *ReturnTypeList* is a subset of the *PassedTypeList*. If the *PassedTypeList* is *NULL* it means that the corresponding element should only return all the types that it could be and there isn't any force for its type. Now let's see how *SemanticCheck()* function utilizes the *TypeCheck()* function. Figure 4-3 shows a simplified version of this function for a *signal assignment statement.*

```
SignalAssignment::SemanticCheck()
{
...

List1 = Target . TypeCheck( NULL );

List2 = RightHandSide . TypeCheck( List1 );
...
}
```

Figure 4-3: SemanticCheck() function for class SignalAssignmeent

As it is seen first the *TypeCheck()* function with the *NULL* parameter for the left hand side of assignment is called. In our example the execution of this function should search in the CHIRE memory representation to find the declaration of the left hand side signal (*"a"*), which in fact will be the memory representation of the statement in Figure 4-4.

```
signal a,b,c,d:bit2;
```

Figure 4-4: Declarations of signals a, b, c and d.

It then will find type *"bit2"* for *"a"* and returns it in *List1*. *List1* is then passed to the *TypeCheck()* function of the right hand side of the assignment and as shown in Figure 3-2, its kind is a "+" operator. In the *TypeCheck()* function implementation of "+" operator the following tasks are done. First all the "+" operator declarations that are visible to this expression are found and those that do not have a return type of *"bit2"* are deleted. In this example there is only one "+" operator that returns *"bit2"*, now the same tasks are done for the left operand "*". These calls finally reach to *"b"* and force it to be the *"bit2"* type. The implementation of *TypeCheck()* function for *"b"* is the same as the left hand side of assignment *"a"* therefore it searches for the declaration of *"b"* and then returns a list containing *"bit2"* to it's caller in the "*" operator. The "*" operator then makes another list like the one made for the left operand and calls the *TypeCheck()* function for its right operand which is an array aggregate. In the implementation of *TypeCheck()* function for aggregates, all the *TypeCheck()* functions for the elements of the aggregate should be called. Here the type *"bit2"* which is an array has been passed to the aggregate therefore the declaration of the type *"bit2"* should be found to identify the type of the sub elements of the array. This is infact the memory representation of code shown in Figure 4-5.

```
type bit2 is array(3 to 2*2) of bit;
```

Figure 4-5: Declaration of array bit2 with static expressions in range type definition part

It is seen that the type of the sub elements of the array is *"bit"* and the size of the array had been declared by two static expressions *"3"* and *"2*2"*. However there are virtual functions that can evaluate the value of a static expression for all the classes that may appear in a static expression. So the size of *"bit2"* is then evaluated by these

functions and the result is *(2*2-3)+1=2*, Therefore this aggregate should exactly have 2 elements and the type of these elements is *"bit"*. In the *TypeCheck()* function of the aggregate, if no error is produced, a list containing type *"bit2"* is returned back to the caller of the *TypeCheck()* function of the aggregate which is the *"*"* operator. The *"*"* operator that have received two lists from it's left and right operands, compares them with the visible *"*"* operator and produces error messages if needed. After this step the *TypeCheck()* function of the *"*"* operator returns a list containing a *"bit2"* to it's caller (*"+"* operator). This *"+"* operator now has finished the call to it's left operand and is ready to call the *TypeCheck()* function for its right operand. Its right operand receives a list containing *"bit2"* and as it is seen in Figure 3-2 it is a function call corresponding to the expression in Figure 4-6.

...+f(g,d=>h)
Figure 4-6: The function call with formal parameter indication of expression in Figure 2-1

Like "*" or "+" operators the *TypeCheck()* function of the function-call searches all the functions named *"f"* which have return type of *"bit2"*. In this example it finds two functions shown in Figure 4-7.

```
function f(a,b:bit) return bit2 is
begin
  return ("00");
end;
function f(c,d:bit) return bit2 is
begin
  return ("11");
end;
```

Figure 4-7: Implementation of the two "f" functions

Therefore it makes a list containing the type list in Figure 4-8a to be passed to *TypeCheck()* function of it's first element *"g"* and a list containing the type list in Figure 4-8b to be passed to *TypeCheck()* function of it's last element *"d=>h"*. The first call of *TypeCheck()* function only will check the type *"bit"* since it doesn't have any *formal part*, but the last call of *TypeCheck()* function should also check the formal name *"d"* of function call to recognize which element of the passed list should be selected. (It is a part of sub program overloading feature of VHDL language that overloading can be done throw the name of the formals and is described in page 25 of VHDL 93 language reference manual [4]).

{(a,"bit"),(c,"bit")}	{(b,"bit"),(d,"bit")}
Figure 4-8: a, b) The Type Lists to be passed to the first and second association parameters of function call in Figure 4-6	

At last this *TypeCheck()* function will return a *"bit2"* to it's caller which is *"+"* operator and then again the *"+"* will return a *"bit2"* to it's caller which is the *SemanticCheck()* function of the *signal assignment statement*. The *signal assignment statement* has received *"bit2"* from it's right hand side that is of course a subset of the list containing *"bit2"*, That it received from its left hand side. Therefore it could be concluded that no error had occurred throw the type checking and the assignment is correct. In complex cases the type lists may contain several type elements and most of them are *MR** type. Using virtual functions, the implementation has become more structured and manageable. If we have a pointer *"P"* of the type of *MR** it's enough to call it's *TypeCheck()* function using *"P->TypeCheck()"* independent of the real type of *"P"*. In the described process of calling *TypeCheck()* functions if the types could not be resolved or matched with the requirements and restrictions an error message with detailed information will be reported to the VHDL programmer, annotating an error in the VHDL input source file.

This example demonstrates how virtual functions in an OOIF can simplify the development of any kind of CAD tools. For example in a simulator different implementation of a *Simulate()* function for different classes can construct the whole execution part of the engine.

4.3. Updating The Pending Pointers

In the described type-check process, whenever a *TypeCheck()* function returns a one member *ReturnTypeList*. The type of the corresponding element is set to that single member of the list. But in cases that there are more than one type element in the list; setting

the type pointer of the element is postponed to the time that a proper decision is possible. In fact whenever a node in the syntax tree can determine its own type, it should call other functions for its children in the tree to force them to set their type to proper values.

5. IMPROVEMENTS AND CONCLUSIONS

We showed how different processing of the context-sensitive grammar of the VHDL is made easier using an OOIF. On the other hand, all VHDL based tools can start from this IF instead of developing their own compiler. The semantic-check algorithm presented in this paper is heavily based on utilizing virtual functions that examine and set the types of different elements. The proposed structure is also useful for developing other CAD applications that have to handle complex constructs.

6. REFERENCES

[1].A.M. Gharehbaghi, M.H. Reshadi, Zainalabedin Navabi *"Intermediate Format Standardization Ambiguities, Deficiencies, Portability issues, Documentation and Improvements"*, Design Automation Conference 2000.

[2]. J.C. Willis, G.D. Peterson, S.L. Gregor, *"TheAdvanced Intermediate Representation withExtensibility / Common Environment (AIRE/CE)"*,IEEE Transaction on Computer, 1998.

[3].AIRE document Version 4.6 at http://www.eda.org/aire/

[4]. IEEE standard VHDL Language Reference Manual, IEEE std. 1076, 1993, The Institute of Electrical and Electronic Engineers, New York, NY.

[5]. Draft IEEE standard Verilog Language Reference Manual, IEEE std. 1364, 1995, The Institute of Electrical and Electronic Engineers, New York, NY.

[6]. M. H. Reshadi, A. M. Gharehbaghi, *"VHDL Intermediate Format Representation"*, CAD Lab. Report 23, University of Tehran, August 1999