Integrating Tools in a VHDL Framework

Juana López, Guillermo Ricalde, Andrés García, Luis Entrena, Juan Goicolea, Serafín Olcoz

Phone: (34) 1-3964832. Fax: (34) 1 3964841

Abstract
This paper shows the main results after the experience of building tools for VHDL applications using some commercial VHDL front-ends. To this purpose we have evaluated several VHDL analyzers for performance, compliance and application support. This work has been done in order to implement tools based in a formal model of VHDL. We stress the necessity of a safe way to comply with the standard avoiding cumbersome interpretations.

1 Introduction

The IEEE standard VHDL is the most widely supported Hardware Description Language (HDL) by the industry, research centers and CAD vendors. The preface of the Language Reference Manual (LRM) of VHDL, [IEEE 88], states that this language is both machine readable and human readable, it supports the development, verification, synthesis, and testing of hardware designs, the communication of hardware data, and the maintenance, modification and procurement of hardware. Because of these features a lot of different tools based on VHDL are emerging to cover all aspects of the electronic design.

VHDL based tools must correctly interpret VHDL source code as defined in the LRM, but problems arise because this definition has not a clear enough interpretation. Problems are so important that some standard interpretations, [IEEE 92] have been given by the IEEE as a result of the VASG (VHDL Analysis and Standardization Group) activity.

A set of tools based in a formal model of VHDL [OLCOc 93] is under development in the FORMAT Project (CEC ESPRIT III project #6128). These tools conform a framework for automated static analysis of VHDL [OLCOa 93] based on Coloured Petri Nets (CPN). We are developing a Petri Net Generator and a Petri Net Analyzer as well as a translator of Petri Nets (PN) into Transition Systems, which is the path to verification tools. Figure 1 shows the architecture of these tools.

A VHDL framework must contain at least an analyzer, an elaborator and a simulator. Tools can be designed from the output of each of them. In our approach the starting point for our verification tools is not the output of the analyzer but the output of the elaborator (figure 1) because the VHDL analyzer does not create any executable model. This model is generated by the VHDL elaborator, being in this way an intermediate step for either simulation or verification tools. We did not find available for our framework a commercial accessible VHDL elaborator tool. We were forced to design our own elaborator from the output of a commercial VHDL analyzer. Therefore, to decide which one to use we tested several analyzers available at TGI: SYNOPSYS, CADENCE, CLSI (now COMPASS) and LEDA.

The results we present in this paper will report on time spent, disk sizes needed by these analyzers, and of course on their standard compliance. The facilities given for building new application tools, such as accessibility of the results of the analysis (data schema) through a procedural interface, will be also reported.

To define this Data Schema, a big effort has been done around the definition of a draft of the standard VHDL.
Intermediate Format (VIF), [BROW 92] and [FONK 92], by the VIFASG. A Schema Definition Language [VIFA 90a], and a Procedural Interface, [VIFA 90b] to manipulate the design libraries has been defined. However, these works are not enough to guarantee the standard compliance, the portability of VHDL descriptions, and the intertool communication that is required in a VHDL framework.

Our proposal will be to extend this schema to contain also the results of the elaboration and include it in the VIF in order to establish a standard base for the integration of VHDL tools (see figure 2). The goal of integrating tools will be to provide a general open framework with standardized interfaces in order to ease the construction and management of integrated open environments, as is proposed by the CFI (CAD Framework Initiative Inc.).

The paper is organized as follows: Section 2 introduces the task performed by the analysis of a VHDL description. Section 2.1 presents the comparison of VHDL analyzers available at IGi, where subsection 2.1.1 shows the results of the time and disk size measurements carried out on the tools and subsection 2.1.2 presents the results of preparing some simple proofs to examine the compliance definition of the analyzers under study. Also section 2.1.3 talks about how to access to the results of the analysis. In section 3 the tool that makes the elaboration of the VHDL description is presented. Section 4 show the conclusions and the appendix shows some code used to check the analyzers.

2 Analysis of a VHDL description.

A VHDL description represents a design hierarchy of modules with strictly controlled interface units (Entities, Configurations, and Packages), and implementation units (Architectures, and Package bodies). The textual description is stored in one or more text files, called design files. The content of a design file can be parsed in order to check the correctness of its syntax. The content of a design file is sequentially organized in one or more design units with some (optional) context clauses, see figure 3.a.

The Analyzer can be considered as a compiler that transforms a design file in a set of interconnected library units in library WORK, see figure 3.b. Each library unit corresponds to a design unit of the design file. The interconnections among them are based on a
The Library System (LS) is a set of design libraries. Each design library is a real "library" where the library units are organized, see figure 3.e. If a design unit is reanalyzed then all library units are potentially affected by such a change, become obsolete and must be reanalyzed before they can be used again. This feature allows the VHDL user to keep a library system up to date. Thus, the analyzer permits design management and reusability of designs.

The output of the analyzer does not create an executable model of a VHDL description. In order to get an executable model, the design entity corresponding to the top of the design hierarchy must be first elaborated. The elaboration of a design hierarchy produces a model that can be executed in order to simulate the design represented by the model.

2.1 Comparison of VIIDL analyzers

We show in this section the results of the comparison made on time spent, disk sizes needed by these analyzers and of their standard compliance.

The comparison made does not pretend to be exhaustive. It refers to some practical aspects such as time needed for the analysis or memory size used, as seen by the users.

The systems under test are: 1- LEDA VHDL System (version 3.0); 2-SYNOPSYS 1076 VHDL Analyzer (version 3.0a-10052); 3-COMPASS (VHDL Tool Integration Platform version 1.5.0.x), and 4-CADENCE (Leapfrog version 1.0).

2.1.1 Performance measurements: We have included the measurements taken about some representative files. These files belong to a SPARC architecture VHDL model developed at TGI. The main comparisons made are related to the time and disk size required to make the analysis because they are the most important performances that we are interested in. We have used the "time" command of UNIX which reports the real time, user time, and system time. The Library size is measured by the number of bytes that the new library unit introduces in the working library.

The system in which measurements were taken is a Sun SPARCstation 2 with the following characteristics: 64 MB (RAM) and 400 MB Internal Hard Disk. We used the /disk directory which had the caharacteristics of table 1.

Figure 4: Real time (in seconds) required in the analysis versus file size in number of characters.

A-First analysis. B-Second analysis.

Figure 5: User time in milliseconds versus file size in number of characters.
The results related to user time are: SYNOPSYS is the fastest, followed by LEDA, CADENCE, and finally COMPASS.

2.1.1.3 System time: We observe a difference in LEDA, between the first analysis and the following ones (see fig 6). We studied the reasons and reach to the conclusion that it is because the measurements were taken working in a disk type mounted on NFS. LEDA is very sensitive to it. SYNOPSYS is the fastest in both analysis. After SYNOPSYS, both first and second time the order is COMPASS, LEDA and finally CADENCE.

The above results also show that time is not a very critical parameter for these tools.

2.1.1.4 Library size: In this graph (see figure 7) we

2.1.1.2 User time: User time comparisons are shown in figure 5. There are no important changes in user time when the size of the working library increases, that is to say, between the first analysis and the following ones.

**Table 1**

<table>
<thead>
<tr>
<th>File system</th>
<th>/disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kbytes</td>
<td>1255494</td>
</tr>
<tr>
<td>Used</td>
<td>1098153</td>
</tr>
<tr>
<td>Avail</td>
<td>31792</td>
</tr>
<tr>
<td>Capacity</td>
<td>87%</td>
</tr>
</tbody>
</table>

This disk is hard mounted, read-write on the NFS (Network File System) from a different SPARCstation.
show the library size in characters for each analyzed file. This feature is important, because it relates to the disk space required at the moment of the analysis, but reliability of the working library is more important indeed. It depends on the method of implementation. The common method consists in a hierarchy of directories and files inside and presents no problems, but the mode of using a single file as working library is dangerous for the consistency of the analysis because if the analysis of one unit fails, all the library will be corrupted.

2.1.2. VHDL compliance: As we have commented before, in 1992 IEEE gave some standard interpretations [IEEE 92] to the LRM [IEEE 88]. They were needed to clarify many aspects of the LRM. Moreover, VHDL International is moving towards the development of a standard VHDL test suite in order to qualify VHDL based tools in the future. We wanted to know what the state of the commercial analyzers with respect to the standard interpretations was. To this purpose, we prepared some examples to check the analyzers. Some results were expected (sometimes the Analyzer should detect the errors, sometimes it should accept some statements). The experiments carried out showed that each tool had a different behavior in certain cases. The complete examples are shown in the appendix., where some statements are in bold type to express that each tool behave in a different manner in their analysis.

2.1.2.1. Experiment 1-aggregates
Some possible cases of aggregates are tested. Most of them are referred to choices (integer discrete ranges and enumerated discrete ranges for choices) and static's (globally static and non-static choices). It is tested that no order in the named association is required, particularly in the case of array aggregates. The five syntactic variants of discrete ranges for choices are also tested.

Checked: The following statement is legal: "(S1,S2,S3,S4) <= BV_4'(1 to 2 =>'0',03=>'1')" where BV_4 is a subtype bit-vector (0 to 3),.., but it is not legal to use the word "others" here: "SN < = bV_N (int_1L2=>'1',others=>'0')" because it is not locally static.

2.1.2.2. Experiment 2 - blocks: We tested some possible cases of the guard signals: different scopes, and implicit and explicit guard signal declarations. Visibility was also tested, particularly the extension to lower level entities through component instantiations as specified in section 9-1 of LRM.

Checked: The explicitly declared signal GUARD can be passed as an actual signal in a component instantiation in order to extend its scope to lower level entities. In one of these lower level entities we tried to use the statement: "s<=guarded '1';" We expected that to be accepted, because it is legal to use a guard signal explicitly as an actual signal in a component instantiation (see 9-1 LRM). In our example the guard signal is an actual in a port association, but ports of any mode are also signals (LRM 4.3.1.2 ).

2.1.2.3. Experiment 3 -attributes: We wrote some attribute specifications. We tried to redefine predefined attributes. We also tried to associate attributes to subelements of objects, directly and through alias declarations. At last, we tried to evaluate attributes of subelements of existing objects. Most of the statements were obviously erroneous to check if the analyzers detected them. References are section 5.1 of LRM and Issue Reports 12, 40, 61 and 64 of VASG ISAC Interpretations.

Checked: It should not be possible to declare an scalar attribute of a subelement of a composite object, although it was through an alias of this subelement.

2.1.2.4. Experiment 4-operators: We wrote expressions using predefined operators and not predefined operators, mainly using integer, real, universal integer and universal real operands. We used also implicit and explicit type conversions. Related Sections of LRM are 7.2 and 7.3.5.

Checked: This line should not be accepted :":"...elsif v1 > 2.1*2 then null;" where v1 is real.

The right hand expression is a product of a universal real and a universal integer (result: universal real). An implicit type conversion for a universal real expression is not defined.

2.1.2.5. Experiment 5 - predefined attributes:
We wrote some references to predefined attributes. Specially, we tested that constraints are checked when using predefined attributes that are functions. Constraint violations in the expression of these predefined attributes should be detected if this expression is locally static.

The format is " T" attribute name (x)" where T must be discrete or physical type or subtype and the parameter x must be an expression whose type is the base type of T. Some analyzers did not detect when the parameter within parenthesis did not belong to the specified type in the prefix.
2.1.2.6. Packages: We have noted that some packages provided by SYNOPSYS are not accepted when analyzing them in other tools. That is a problem for the portability of the designs. These packages are not right because they use the reserved word "others" in a choice not included in the list of valid context of section 7.3.2.2. of the LRM.

We have found that each tool interprets VHDL in a different manner. In the experiments made the answers of the analyzers were different. This is very dangerous because we need a real standard. Several validation test suites of the LRM have been proposed, [BERM 89], [ARM 89], [SERA 90], [SCOT 90], [ARMS 90] and actually VHDL International has asked to the VHDL Technology Group to develop a standard VHDL test suite. This would be a great improvement for objectivity and independence of the VHDL tools.

2.1.3 VHDL Data Schema and Procedural Interface:

The VHDL Analyzer is the platform to build the framework of VHDL based tools. The result of the analysis is usually stored in an object-oriented data base which is accessed through a programming interface. There has been an effort by the VIFASG to standardize the format used to store the results of the analysis (VIF) and the access to the data base, but this standard is not complied by all commercial analyzers.

A standard VHDL intermediate format is strongly desired in order to ensure the portability of designs and the compatibility of new tools.

Only two of the analyzers studied for this work, namely LEDA and COMPASS, provide a procedural interface. The COMPASS analyzer was the first commercial tool that gave user access to a data base containing the results of the analysis. This tool was released before the definition of the VIF. The LEDA analyzer appeared later and is based on the VIF proposal. Moreover the LEDA analyzer incorporates some additional features that make easier the task of building VHDL applications. This features include the possibility of extending the basic schema to store user-defined data.

3. Elaboration of a VHDL description.

The elaboration is the previous step to execute a VHDL description as the LRM Standard [IEEE 88] states. Elaboration transforms a design hierarchy into a flat model consisting of processes interconnected by a network of signals, and their associated information, [OLCOb 93], see figure 8. This model can be considered a set of processes because there is no hierarchy among these processes. Elaborated processes are composed of VHDL sequential statements that are cyclically executed. The simulation of this model is done by an external agent, the kernel process. Although the elaborated model is flat, each process can be seen as a hierarchy of statements grouped in subprograms (procedures) like in any other structured programming language.

The elaborated model does not depend on the description style (behavioral, structural, data-flow, or mixed), chosen by the designer, nor on the abstraction level of the VHDL description. This model contains the same information that the original VHDL description. Although it is a flat model, the information of the design hierarchy can be recovered through the signals. VHDL simulators usually offer the possibility of debugging VHDL source code line by line because they retain links between the elaborated model and the design files corresponding to the VHDL description. This is possible because the VHDL elaborator saves the needed information to perform the backtracking from the elaborated model to the VHDL source code.

We studied different possibilities to implement the tool that performs the elaboration task. If a common standard, defining the intermediate format and a common procedural interface that allowed the access to the database had been available (as is proposed in the VIF), the way to make the tool would have been the one presented in figure 9.A
Actually, the different CAD vendors that allow access to the analyzed model have their own schema definition and procedural interface. If a tool developer wants to have a front-end that admits different analyzer outputs as inputs, he/she cannot use a common design. The design must be split into two parts: database dependent routines for each input, and common database independent routines (figure 9.A). This is not a good solution, because it forces the developer to write new code for each analyzer. The Information model of the elaborated model should be unique. The way this information model is implemented depends on the input database. There are three possibilities. In the first possibility the elaborator writes its output in the same database used by the analyzer and the schema can be augmented. This possibility has few problems. In the second possibility the schema can not be augmented and the output is still written in the same database. It is very difficult however to put new data if we cannot change the schema. In the third possibility the output is a different database. It is, however a very hard work to translate all the useful data of the input to the output in the case of a complex tool like an elaborator.

A similar problem appears when different tools define their own information model for the elaborated model. Since the elaboration is a prerequisite for simulation, as well as for the application of formal methods to VHDL, a standard schema definition for the elaborated model would be very valuable, in order to eliminate writing database dependent routines. Also, the availability of commercial tools that provide an interface to the elaborated model would be very helpful to the development of new applications in the above mentioned domains. In that case, all the tools of the proposed framework could work on the same database, using the same procedural interface. These ideas are in accordance with the directives of CFI, in order to integrate tools in a framework, with the objective of providing an open environment.

4 Conclusions.

We have reported the steps taken in order to integrate new applications into a VHDL framework, and stated the problems we have found. Measurements have been taken to check the VHDL Front-end of the framework, the analyzer. With respect to the time and memory required by the commercial analyzers, measurements indicate there are no significant problems. Only two of the commercial analyzers available provide a procedural interface to access to the database. When checking the compliance of the commercial tools with respect to the VHDL LRM, problems arise due to the ambiguity of this standard. In spite of efforts for standard interpretations [IEEE 92], they are not enough for a standard definition. The effort made to establish a standard VIF is recognized but only some guidelines

Figure 9. Design method: A.-Desired way of designing the tool. B-Final design.
have been produced so far and it would be very good to have a standard IEEE VIF. A standard Validation Test Suite definition is needed to ensure the compliance of the tools appearing in the market. We also propose to make a similar effort to define a standard intermediate format for the result of the elaboration, as many applications can be developed starting from the elaborated model. (For example, the elaborator developed is being used in fact as the front-end for verification tools and can be the starting point for some other VHDL tools in the near future).

However, the first problem to be solved is the lack of a formal syntax and semantic definition of the standard VHDL. In this sense, projects that define a formal semantic like the FORMAT project will serve to advance towards the solution of these problems.

Appendix- Examples of the code used to check VHDL compliance.

This appendix shows the VHDL code of some files used to check VHDL compliance.

Experiment 1

ENTITY aggregates IS GENERIC ( N : Integer := 5 );
END;
ARCHITECTURE a OF aggregates IS
  subtype Bv_4 is Bit_Vector ( 0 to 3 );
  subtype Bv_N is Bit_Vector ( 1 to N );
  subtype Int_1_2 is Integer range 1 to 2;
  signal s1,s2,s3,s4 : Bit ;
  signal sN : Bv_N;
  signal i : Integer ;
BEGIN
  ( s1,s2,s3,s4 ) <= Bv_4'( 1 to 2 =>'0', 03 => '1' ) ;
  ( s1,s2,s3,s4 ) <= Bv_4'( others=>'1' ) ;
  ( s1,s2,s3,s4 ) <= "101" and B"101" and o"5" ;
  sN <= Bv_N'( others => '1' ) and ( N => '0' ) ;
  sN <= Bv_N'( Int_1_2 => '1', others => '0' ) ;
  sN <= Bv_N'( Int_1_2 => '1' ) ;
  sN <= Bv_N'( Int_1_2 range 1 to 1 => '1' ) ;
  sN <= ( Int_1_2 => '0' ) ;
  sN <= ( 2 to N => '0' ) ;
END;

Experiment 2

ENTITY guarded_entity IS PORT ( GUARD : INOUT Boolean );
END;
ARCHITECTURE a OF guarded_entity IS

signal s:Bit;
procedure change(signal g:inout Boolean ) is
begin
  g <= not g ;
end;
BEGIN
change(guard);
 s<=guarded '1';
END;

ENTITY guarded_blocks IS
END;
ARCHITECTURE a OF guarded_blocks IS
signal s, t, u : Bit;
component comp port ( Guard_Port : inout Boolean ) ;
end component;
for all-comp use entity
  work.guarded_entity(a) port map(Guard_Port) ;
BEGIN
  b1:block
  signal GUARD : Boolean;
  begin
  s <= guarded '1';
  b2:block
  begin
    t <= guarded '1';
  end block;
  b3:block ((s or t) = '0')
  begin
    u <= guarded '1';
  end block;
  C4:comp port map ( GUARD ) ;
end block:
END;

Experiment 3

ENTITY attributes IS END;
ARCHITECTURE a OF attributes IS
attribute usr attr:Integer;
attribute usr attr of a:architecture is ((2)*(3));
constant c:Bit Vector(0 to 1):="01";
alias c0:Bit is c(0);
alias c1:Bit is c(1);
alias c2:Bit_Vector(0 to 1) is c;
attribute single0:Boolean;
attribute single0 of c0 : constant is true;
attribute single0 of c1 : constant is true;
type Boolean2 is array(0 to 1) of Boolean;
attribute agg: Boolean2 ;
attribute agg of c : constant is (False, True);
signal s : Integer ;
signal t : Boolean2 ;
BEGIN
s <= a'usr_attr ;
t <= (c(0)'single0 & c1'single1 ) or (c0'agg & c2(1)'agg ) ;
t <= c0'agg & c2(1)'agg ;
t <= c0'agg & c'agg(1) ;
t <= c0'agg ;
t <= c2'agg(1) & c'agg(0) ;
END;

Experiment 4
ENTITY arithmetic IS END;
ARCHITECTURE a OF arithmetic IS BEGIN
process
  variable i1,i2 : Integer;
  variable r1,r2 : Real ;
begins
  r1:=+2.3 ;
  r2:=real(-2) ;
  i1:=3 ;
  r2:=r2*r1 ;
  r1:=i1 ;
  r2:=r1 * 3.1 ;
  r2:=i1 * 3 ;
  r1:= real (2.3 * 3) ;
  r2:= real(i1) ;
  i1:= integer(r1) ;
  i2:= integer (4.6 / 2) ;
  elsif r1 >=2.3*1.7 then null ;
    elsif r1 > 2 then null ;
  end if ;
  wait ;
end process ;
END;

Experiment 5
ENTITY pred_attributes IS END;
ARCHITECTURE a OF pred_attributes IS type New_Int is range 20 downto 10 ;
subtype Restr_Int is New_Int range 15 downto 12 ;
constant c : Restr_Int := Restr_Int 'Val(1) ;
signal t : Integer ;
signal b : Boolean ;
BEGIN
  t <= New_Int'Pos(10) ;
  t <= Restr_Int 'Pos(15) ;
  b <= 't'Stable(10 ns)'Delayed(5 ns)'Quiet ;
END;

References.


