

OSYS: Tools for behavioral synthesis of ICs with VHDL.

J Benzakki*, D. Conan*, M. Israël**

***Cedric/IE 18 Allée Jean Rostand, 91002 Evry Cedex, France**

Email: jbenz@cnam.cnam.fr

**** Université d'Evry Val d'Essonne Bd Des Coquibus 91025 Evry Cedex, France**

1 Abstract

In [DON] P. Beauvillard from Cadence Design Systems says : *"Even if it were possible to design a complex ASIC at the gate level, statistics show that about half the time it won't work when interfaced to the rest of the system. It's imperative to simulate performance at a high level of abstraction. You need an HDL to do this"*.

This is quite true, that is why the development of rapid prototyping environments working from a behavioral specification including genericity and abstract data types [WHI, BIF] is important. The methodology we use is inspired by the Balzer's model, used in software engineering [BAL]. As it is illustrated by Figure 1, the user will be able to simulate his description at each phases of the synthesis process and refine his specifications depending on the results of the simulation.

This paper will describe the first phase of this project : the implementation of genericity and abstract data types in VHDL, the inclusion of VHDL in the interactive programming environment generator : Centaur, and the generation of full behavioral VHDL from generic VHDL.

2 Introduction

Today, the main features needed for IC design environments are very similar to the ones of software engineering. The designer has to :

- manipulate different formalisms corresponding to the different levels of abstraction. Thus language parametrized tools should be interesting.
- specify semantic computations on the preceding formalisms such as :
 - analyse the high level description in order to extract the resources needed,
 - transform the high level description in another lower level formalism (from a VHDL behavioral description to a Data-flow one). Thus it should be interesting to have the ability to generate automatically tools from the semantic specifications.

In order to realize the semantic analysis the input program should be represented by an internal format easy to manipulate. In software engineering the most used internal representation is called Abstract Syntax Tree (AST).

All these functionalities can be provided by Centaur [CLE] which:

- is language independent. And therefore can be parametrized according to the user specification of the language,
- specifies the syntax using two formalisms : Metal from text to AST and PPML from AST to text and the semantics by inference rules written using the Typol language (Prolog like). These rules will operate by induction on the abstract syntax,
- offers a sophisticated user interface.

Until now CENTAUR has been used mainly for software engineering, it can handle languages such as Pascal, Ada, and prototyping languages (Setl). It has been used to translate prototyping languages to optimized Ada.

Centaur, represents the kernel of the high level synthesis system we are developing (Figure 1). VHDL being, at this time the entry language to which we add genericity and abstract data types (see [WHI]). In parallel of this development, work is done on resources allocation and formal specification language.

3 CENTAUR and the VHDL grammar (the syntax problems)

Figure 1 gives the overall scheme of a high level synthesis environment which generates a Data Flow VHDL description of a circuit.

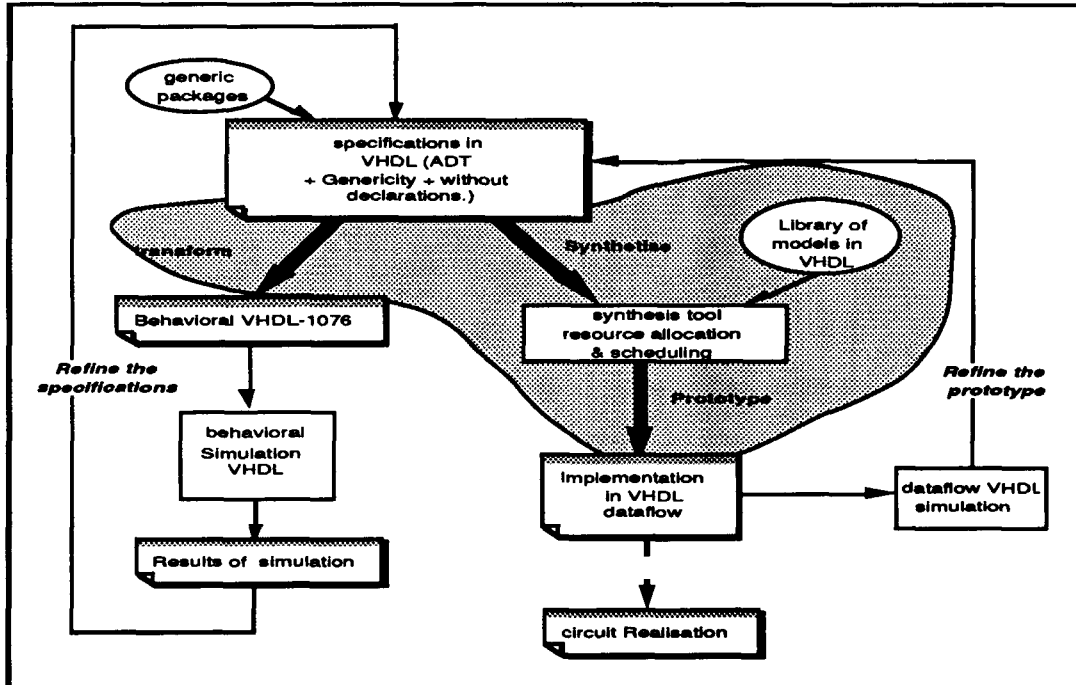


FIGURE 1 : The process from a VHDL high level specification to circuit implementation. The shaded portion corresponds to the use of Centaur : transform uses the process described in Figure 1, it is the first step, then synthesize make resources allocation and scheduling according to a cell library and prototype transform the AST into a dataflow VHDL description. This process follows the Balzer model [BAL].

Figure 2 shows the different phases through Centaur.

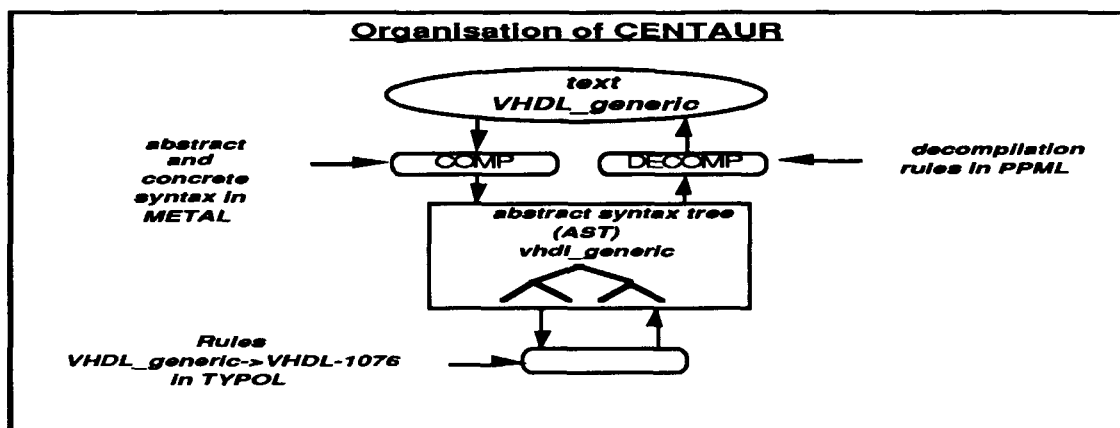


FIGURE 2 : The Centaur overall organization showing the different steps to follow in order to add a new language with Centaur: give the syntax rules in METAL, give the semantics rules in Typol and give the decompilation rules in PPML.

As we know, VHDL has been developed using a number of Ada constructs and, therefore, it is quite a verbose language, leading to quite long descriptions. Thus we had to make some choices for its inclusion into Centaur. In the remaining of this section we will present, with the implementation of genericity in VHDL, the different problems we had to face and the choices we have made.

3.1 The METAL description

As we have seen in Figure 2, before including a new language into Centaur, we have to describe its concrete and abstract syntax with the METAL (META Language) formalism [KHA]. This formalism allows to describe a grammar by specifying its concrete and abstract syntax. Once this is done, Centaur produces, according to the syntax entered, an analyzer, a tree generator and a decompiler allowing the manipulation of VHDL files :

- the analyzer transforms a VHDL text into an Abstract Syntax Tree (AST),
- the decompiler generates a VHDL program from an AST, the rules are given in the PPML language [CEN] (see section 3.4).

3.2 Problems due to the VHDL standard, choices

The Centaur system uses YACC, so the grammar rules must be given in a BNF format and the VHDL grammar must be LALR(1), which is not the case in the standard. Thus we had to generate a LALR(1) VHDL grammar. Then, in view of the size of the grammar, we entered the rules chapter by chapter and resolved the conflicts as we progressed. The only restrictions we have made are for the paragraphs "association_element" and "name". The simplification were done according R. Airiau [AIR] and S. Datta [DAT].

Then, we had to make the following choice :

Shall we define a new abstract syntax according to the concrete syntax structure, or shall we group the semantic concepts under the same operators ?

It is difficult to define a deep abstract syntax for a language such as VHDL because it has not been designed under this constraint, and more, the target file must be in VHDL-1076.

Thus, we have defined an abstract syntax edition oriented. In the future we might have to define a second formalism, as it is suggested and applied by the designers of Typol [DEY].

3.3 Introduction of genericity

a) Concrete syntax :

In order to illustrate the inclusion of the VHDL grammar, we present, in Table 1, the concrete syntax of abstract data types [BFI] and genericity.

<model_declaration>	::=	"model" <generic_type_part> <object_subprogram_declaration_part> <subprogram_declaration> "end" "model" ";";
<object_subprogram_declaration>	::=	"with" <subprogram_specification> "is" "<>" ";";
<generic_subprogram_instantiation>	::=	"procedure" <designator> "is" "instance" <identifier> "(" <association_list> ")" ";";
<generic_subprogram_instantiation>	::=	"function" <designator> "is" "instance" <identifier> "(" <association_list> ")" ";";

Table 1 : Part of the concrete syntax for a generic model declaration. The operator model declaration in bold has an arity of 3.

b) Abstract syntax

The set of abstract syntax rules characterizes the tree structure of a language. An Abstract Syntax Tree (AST), is associated to each program with tags corresponding to the AST operators. Each operator has a fixed or variable arity (number of sons). For each son, all the possible root operators are indicated, such a structure is called a "phylum".

For example, in Table 1, the operator "model declaration" has an arity of 3 — 3 phyla — which types are:

- <generic_type_part>
- <object_subprogram_declaration_part>
- <subprogram_declaration>

Then, the corresponding phyla of the operator "model declaration" is :

```
model_declion->  GENERIC_TYPE_PART
                  OBJECT_SUBPROGRAM_DECLARATION_PART
                  SUBPROGRAM_DECLARATION.
```

Each one of these phyla (in capital) could be defined in the same way.

Because many syntaxes can represent the same language, writing an abstract syntax is more subtle than just listing the operators and phyla. Let us present two examples.

a) The designer, wants an abstract syntax for edition (i.e. decompilation). So, he chooses a formalism which sticks to the concrete syntax. But this may lead to redundancy. Let us take the example of generic subprograms instantiation. A first solution could be :

```
<generic_subprogram_instantiation> ::= "procedure" <designator> "is" "instance"
                                     <identifier> "(" <association_list> ")" ";" ;
                                     generic_subprogram_instantiation(atomic_node-atom('procedure'),
                                     <designator>, <identifier>, <association_list>)
<generic_subprogram_instantiation> ::= "procedure" <designator> "is" "instance"
                                     <identifier> "(" <association_list> ")" ";" ;
                                     generic_subprogram_instantiation(atomic_node-atom('function'),
                                     <designator>, <identifier>, <association_list>)
```

An action is associated (in italic) to each BNF rule, indicating which AST should be build with this BNF rule during the syntax analysis. Such an organization of the syntax allows PPML to choose between procedure or function instantiation because of the existence of the *atomic_mode* operator and operand.

b) The designer wants to make computation on the semantic, he has to extract the maximum of information from each node. the syntax is then more compact but it is impossible to find a correct structure for editing because some syntax elements have been merged and then are difficult to retrieve. This leads to the following implementation :

```
<generic_subprogram_instantiation> ::= "procedure" <designator> "is" "instance"
                                     <identifier> "(" <association_list> ")" ";" ;
                                     generic_subprogram_instantiation(<designator>, <identifier>,
                                     <association_list>)
<generic_subprogram_instantiation> ::= "procedure" <designator> "is" "instance"
                                     <identifier> "(" <association_list> ")" ";" ;
                                     generic_subprogram_instantiation(<designator>, <identifier>, <association_list>)
```

In this example, the abstract syntax is simplest, but PPML cannot express the difference between the two instantiations procedure and function which have been merged, and the generation of a VHDL program is impossible.

The abstract syntax build for genericity is presented in Table 2. All the corresponding chapter is presented.

<pre> abstract syntax MODEL_DECLION ::= model_declion; model_declion -> GENERIC_TYPE_PART OBJECT_SUBPROGRAM_DECLARATION_PART SUBPROGRAM_DECLARATION; GENERIC_TYPE_PART ::= generic_type_part; generic_type_part -> GENERIC_TYPE_DECLARATION +...; GENERIC_TYPE_DECLARATION ::= FULL_TYPE_DECLARATION OBJECT_TYPE_DECLARATION; OBJECT_TYPE_DECLARATION ::= objet_type_declaration; object_type_declaration -> IDENTIFIER OBJECT_TYPE_DEFINITION; OBJECT_TYPE_DEFINITION ::= atomic_node; atomic_node -> implemented as IDENTIFIER; OBJECT_SUBPROGRAM_DECLARATION_PART ::= object_subprogram_declaration_part; object_subprogram_declaration_part -> OBJECT_SUBPROGRAM_DECLARATION *...; OBJECT_SUBPROGRAM_DECLARATION ::= object_subprogram_declaration1 object_subprogram_declaration2; object_subprogram_declaration1 -> SUBPROGRAM_SPECIF; object_subprogram_declaration2 -> SUBPROGRAM_SPECIF; GENERIC_SUBPROGRAM_INSTANCIATION ::= generic_subprogram_instantiation; genric_subprogram_instantiation -> ATOMIC_NODE DESIGNATOR IDENT ASSOCIATION_LIST; ATOMIC_NODE ::= atomic_node; end chapter; </pre>
<p>Table 2 : Full chapter corresponding to the inclusion of genericity in VHDL. The first part is the concrete syntax and the second one the abstract syntax.</p>

3.4 PPML: an example

In Table 3 we give an example of a VHDL program using the genericity and abstract data types, its symbolic AST representation as specified by Centaur is represented in Figure 3.

PPML works on the AST shown in Figure 3, we have to describe how the tree should be translated into a VHDL text. We have obtained a VHDL text with spacing, indentation, fonts we wanted. If, for a given formalism, the editing program has not been built (i.e. for an operator or for a branch in the AST) Centaur applies a generic default program which will build a VHDL construct.

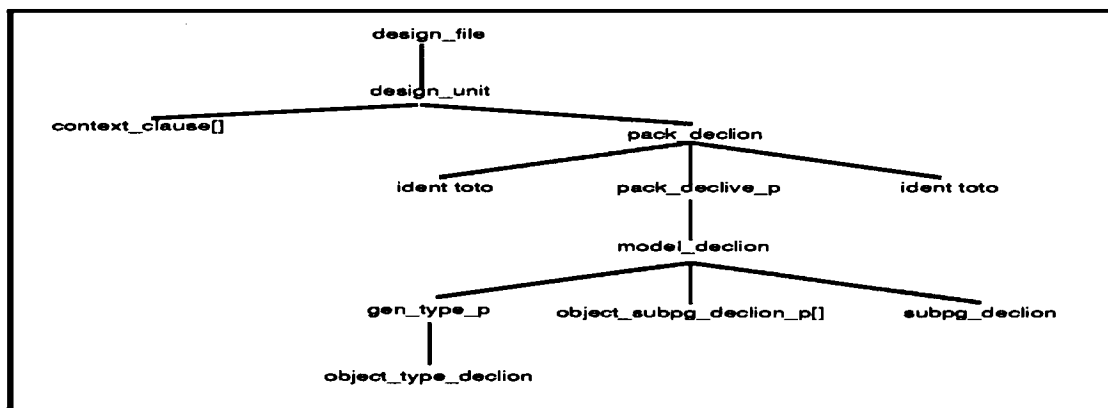


FIGURE 3 : AST representation corresponding to the grammar described in Table 2. [] means an empty list.

4. Typol and Semantic rules applied to genericity

After having implemented all the VHDL language into Centaur with abstract data types and genericity as presented in the preceding sections we still have a problem to resolve : there is no VHDL compiler available handling the genericity ! Therefore, we will present how to generate a full VHDL-1076 program from a Generic VHDL program. First we have to define the semantic specifications. In Centaur these specifications have to be described in Typol (see Figure 2).

A method for specifying the semantic aspects of languages is included in Centaur, so the system is not restricted to manipulations based on syntax. This section gives an overview of Typol language and its compiler, and shows how it is possible to take advantage of semantic descriptions in an interactive environment for our example in adding abstract data types and genericity to VHDL.

4.1 An introduction to semantic specifications with Typol

Typol is an implementation of Natural semantics [KHA]. It can be used to specify static semantics, dynamic semantics, and translations. A Typol specification embodies very intuitive notions. For example to specify the static semantics of a PASCAL like language, we could formalize the type-checking of declarations, instructions, and expressions. Clearly the type-checking of the declaration part "elaborates" an environment containing the bindings of variables to their types. This is represented by a Typol *judgement* of the form :

$$I \vdash D : t$$

where the variable D has the type of the declaration and the variable t has the type of the environment. Similarly, given an environment the type-checking of an expression "returns" the type of that expression, which is written in Typol under the form :

$$t \mid \vdash E : t'.$$

Finally, instructions have no type but they must be "well typed" with respect to an environment, $t \mid \vdash D$.

Of course , it is necessary to describe all the above mentioned predicates, "elaborate", "return", and "well-typed". A Typol program is roughly an unordered collection of axioms and rules of inference. Following is a typical axiom to express that the type of an identifier is found in some typing environment :

$$\{ ID : t' \}. t \mid \vdash ID : t'$$

A rule has basically two parts, a numerator and a denominator. These can contain variables that allow the rule to be instantiated (as PROLOG's variables). The numerator is an unordered collection of equations, the premises of the rule. Intuitively, if all premises hold, then the denominator, a single equation, holds. A typical rule is shown below :

$$\frac{s \mid \vdash ID : t' \ \& \ s \mid \vdash EXP : t'}{s \mid \vdash ID := EXP}$$

This is the static semantics rule for an assignment $ID := EXP$ (as it has been described in the abstract syntax). It can be used to "prove" that an assignment is "well-typed" within a given environment by proving that the identifier (left side) has type t' and that the expression (right side) has the same type. In other words, to answer the query, one has to provide a proof tree using semantic rules.

4.2 Compiling Typol

The Typol compiler generates Prolog code. To use this code from Centaur, a Prolog query, must be constructed and mailed to the prolog engine for resolution. The debugging of Typol programs is driven by the underlying Prolog evaluation

mechanism which uses a depth first and left to right strategy. In other words, to prove the conclusion of a rule one tries to prove the premises from left to right until there is no more premise to prove. To control precisely the Prolog execution from Centaur, we communicate via messages with the prolog engine. There are four possible messages : try (a rule), proved(a rule), back(to backtrack over a rule), or fail(to prove a rule). The basic replies we give are : continue, fail (to force backtracking), retry, and abort [CLE].

4.3 Application to genericity

We will illustrate the use of Typol on the permutation program presented in Table 3. Such a program cannot be analyzed by a VHDL compiler. Table 4 shows the full VHDL program generated after the Typol analysis. This program respects the VHDL-1076 standard.

<pre> package toto is --model --type elem is private; --procedure perm(first,last: inout elem); --end model; end toto; package body toto is --procedure perm(first,last: inout elem) is --variable temp:elem; -- begin -- temp:=first; -- first:=last; -- last:=temp; -- end perm; end toto; entity test is end test; </pre>	<pre> architecture generic of test is --procedure integer_perm is instance perm(integer); procedure perm(first,last: inout integer) is variable temp:integer; begin temp:=first; first:=last; last:=temp; end perm; begin process variable x,y:integer; begin --integer_perm(x,y); perm(x,y); wait for 30 ns; end process; end generic; </pre>
<p>Table 3 generic program VHDL_1076's file generated after Typol's execution. This file does no more contain genericity and ADT, it can be enter to a full-vhdl compiler.</p>	

The algorithm used to generate such a file must go through all the compilation units in order to extract :

- the "model" declarations and the generic subprograms instantiations which have to be transformed into comments,
- the subprograms bodies which have to be included in the "package body".

These different elements been grouped, we have to establish the correspondance between the instantiations and their models and, between the subprograms and their models. Then, using the instantiated types, we can create a new tree for the subprograms bodies.

With this algorithm we have to manipulate new concepts, we must have :

- a phylum which will be a declaration list of "model",
- a phylum which will be an instantiation list and,
- a phylum which will be a subprogram bodies list.

These new concepts are defined in a new abstract syntax.

4.5 Inference rule example

After verifying all the correspondences described in the preceding section, we must define a rule which will replace each generic subprogram instantiation by the subprogram body with instantiated types. The rule is then :

$equality(INSTANCE, MODEL) \&vefif_type(effective\ parameters, formal\ parameters)$
 $\vdash procedure\ P1\ is\ instance\ P\ (effective\ parameters) \rightarrow procedure\ P\ (formal\ parameters)\ is\ deds\ beg\ in\ instance,$

The meaning of this rule is the following one :

The instance P1 of the generic subprogram P will be replaced by the subprogram body P and its list of formal parameters using non generic types if there is a correspondence between the instantiation and the model and, type matching between the list of effective parameters and the list of formal parameters.

5. Conclusion

Using HDLs implies a different approach than the one used in the past for DA environments. Until recently, IC designers could be compared as assembly programmers. The situation is evolving and, most of the designers have more experience in software. Thus, design automation tools designers are faced with the same problems than software ones. That is why we have to be aware of what is being done in software engineering and see what could be applied to Design Automation tools.

We have presented the first part of our system, the use of Centaur has been very helpful and well understood by the programmers. For example, it took 2 months for a student, in a training project, to enter the VHDL grammar into Centaur (he did not know Centaur before).

This first part of the work is very encouraging and we have settled a research team between software engineering and design automation researchers.

The next steps will be the development of generic resources allocation and scheduling and generation of VHDL Data-flow. We are also studying the constructs needed to build a specification language on VHDL.

6. References

- [AIR] : R. Airlau, J.-M. Bergé, V. Olive, J. Rouillard: *"VHDL du langage à la modélisation"*, Presses Polytechniques et Universitaires Romandes, 1990.
- [BAL] : R. Balzer "A 15 year perspective on automatic programming", IEEE transactions on Software Engineering, Nov 85, p1257.
- [BIF] : M. Israel, J. Benzakki, M. Francois "Introduction of Abstract types and genericity in VHDL": , Poster Presentation for Euro-VHDL'91 at Stockholm
- [BIS]: J. Benzakki M. Israel : "Software Engineering in High Level Synthesis" :VHDL-FORUM for CAD in Europe, Spring'91 Meeting(Marseille-IMT)
- [BOR]: P. Borrás, D. Clément & al. "CENTAUR, The system" : RR INRIA n° 777, Dec 87
- [CEN] : Centaur version 1.0, reference manual.
- [CLE] D. Clément : "GIPE : Generation of Interactive Programming Environment". From TSI, Vol 9 n° 2 -1990- p 157
- [DAT] : S. Datta "IEEE-1076 VHDL grammar, Yacc & Bison format" Air Force Inst., University of Cincinnati, Nov 30, 1987.
- [DEY] : T. Despeyroux: "Executable Specification of Static Semantics", Semantics of Data Types, Lecture Notes in Computer Science, Vol. 173, Springer - Verlag, Juin 1984.
- [DON] : Mike Donlin "Asic complexity fuels drive to HDL design" :, Computer Design , May 91
- [KHA 87] : G. Khan : "Natural semantics", Proceedings of STACS 1987, Lecture Notes in Computer Science, Vol. 247, Springer-Verlag, March 1987
- [KHA]: G. Khan, B. Lang, B. Mélése: "METAL: a formalism to specify formalisms", Science of Computer Programming, Vol. 3, pp. 151-188, North-Holland, 1983.
- [WHI] : G.S Whitcomb, A.R Newton; "Abstract Data Types and High-level Synthesis" : 27th DAC 1990, p-680