DESIGN OF REUSABLE VHDL COMPONENTS USING EXTERNAL FUNCTIONS

Vytautas STUIKYS

Kaunas University of Technology
Studentu St. 50, 3031 Kaunas, Lithuania, E-mail : vytas.stuikys@if.ktu.lt

Abstract. This paper presents a method how to represent and build a reusable VHDL component. The representation is based on external functions use. The design procedure is described as transferring of characteristics intrinsic for a given class of domain objects and features from a given VHDL model by re-coding and extending this model with new features. The formal syntax of the functions is given. The component instantiation is performed via pre-processing.
Key words: reusable component, template, VHDL model, pre-processing, external functions

Introduction

Our aim is to work out a framework for the development of reuse templates, as building blocks for a virtual reuse library. In the context of this paper a *template* is defined as a collection of symbols that constitute strings and they, in turn, are combined together to form a *program-like text*. The main feature of this text is that it contains the text *fragments* of two languages: a *basic* language (by this we mean VHDL, the hardware description language[2]) and *metalanguag*e. A role of the metalanguage is to describe that part of the basic program which is to be generated by an external system, *a pre-processor*. So, the text of the template consists of the VHDL fragments and fragments defined by a metalanguage, but the latter fragments must be built-in into the text according to the syntactic rules of the basic language. A fragment may be either a complete structure of the basic program or its constituent.

Semantically, using a metalanguage as an external mechanism, we can add *new features* to the basic program and extend its capability and to enhance reusability. That extension of the initial program enriched by new features we call a *reuse* template. So, in the reuse template we may apply and implement not only those reuse capabilities which are supported by the basic language (for example, generic and packages of VHDL), but as well to introduce the new ones using *external reuse mechanism.*

The concept of pre-processing and template building is not new in the field of software reuse. Many different aspects of this problem can be found in numerous publications [4-6,9,12-14]. Reuse related with VHDL is considered in [1,3,7,8,10]. Our contribution is the representation of reuse templates based on the external *functions* use as well as a framework for the *reusable component* design (by this term we mean a reuse template plus user interface for that template instantiation).

The main task of the discussion delivered in this paper is to show how external mechanism works in detail and what is the extent of its capabilities. In the next

Section we will introduce a formal syntax of the external functions. The rest Sections are dedicated for the description of the reusable component design.

A formal syntax of external functions

We will introduce a list of functions called external or template functions. Each function has one or more arguments but the returned value of a function is always a string. We will not make a distinction between a string of digits or any other symbol. Also no restrictions on the length of the string will be introduced. A string might be either a part of a statement or the entire construct of the program text to be generated. To define a function , we use the following notation:

a pair of symbols "<>"-is used to enclose the construct to be defined; "::="is a definition symbol; a vertical bar "|" separates alternative items; the double pair of braces "{{}}" is used to enclose optional items.

The above mentioned symbols are symbols of a *metametalanguage* because the language of template functions is a *metalanguage* with respect to the basic language. The alphabet (or symbols) of the metalanguage is as follows.

<*metalanguage_symbols*>::=@| {| }| [|]|<*vhdl_symbols*> .

A single pair of braces "{}" is used to enclose the VHDL constructs. Square brackets "[ ]" enclose the argument list of a template function; @ is a special mark to denote the beginning of each template function.

<*template_function_definition*>::=<*beginning*><*function_name*>
[<*argument_list*>]

<*beginning*>::=@

<*function_name*>::=<*short_notation*>|<*long_notation*>

<*short_notation*>::=***sub|b2d|d2b|gen|if|for|con|case***

<*long_notation*>::=***substitution| binary_to_decimal | decimal_to_binary***
| generation | branch| for_loop | concatenation | case

Note that the end of a function is defined by the context, i.e. by the symbol "]", the end of the argument list. The functions are considered in the order of a number of arguments they have. So, we discuss first the functions with one argument. The argument list <*argument_list*> will be defined separately for each function.

One-argument -functions.

<*function_sub*>::=@***sub***[<*arg*>]

<*arg*>*::=*<*template_constant*>|<*template_variable*>|<*template_expression*>

<*template_constant*>::=<*template_designer_specified_natural* >|<*vhdl_string*>

<*template_variable*>::=<*user_specified_name*>

<*user_specified_name*>::=<*name*>

<*template_expression*>::=<*factor*><*arithmetic_operator*><*factor*>

|[<*factor*><*arithmetic_operator*><*factor*>]

|<*template_expression*><*arithmetic_operator*><*factor*>

|<*factor*><*arithmetic_operator*><*template_expression*>

<*factor*>::=<*natural*>|<*template_variable*>

<*arithmetic_operator*>::= +| – | *| / | ^

<*function_sub*_value>::=<*template_variable_value*>|<*template_expression_value*>

|<*template_constant_value*>

<*template_expression_value*>::=<*natural*>

<*template_variable_value*>::=<*natural*>

<*natural*>*::=*<*natural_digit*>|<*natural*><*natural_digit*>

<*natural_digit*>::=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

For example, let be k:=5 ; then @***sub***[k]=5 @***sub***[2^k]=32.

<*function_b2d*>::=@***b2d***[<*binary_of_natural*>]

<*binary_digit*>::=0|1

<*binary_of_natural*>::=<*binary_digit*>|<*binary_of_natural*><*binary_digit*>

<*function_b2d*_value>::=<*natural*>

For example, @***b2d***[101]=5 .

Two-argument-function.

<*function_d2b*>::=@***d2b*** [<*list_item*>,<*list_item*>]

<*list_item*>::=<*template_variable*>|<*template_expression*>

The first argument means decimal item to be converted to binary, while the second one presents a number of binary digits.

For example, let *k*:=5; *r*:=3 ; then @***d2b***[*k,r*+1]=0101.

IF *k*:=5; *r*:=2 ; then @***d2b***[*k*,*r*+1]=101.

Three-argument-function

<*function_if*>::=@***if***[<*arg1*>,<*arg2*>{{,<*arg3*>}}]

<*arg1*>::=<*logic_expression*>

<*logic_expression*>::=<*template_variable*><*relation_operator*><*natural*>

|<*template_variable*><*relation_operator*><*template_expression*>|

<*template_expression*><*relation_operator*><*template_expression*>

<*relation_operator*>::= < |>| = |<= |>=| /=

<*logic_expression_value*>::=0|1|**true**|**false**

<*arg2*>::={<*vhdl_string*>}|<*template_function_definition*>

|{<*vhdl_string*><*template_function_definition*>}

|{<*template_function_definition*><*vhdl_string*>}

<*arg3*>::={<*vhdl_string*>}|<*template_function_definition*>

|{<*vhdl_string*><*template_function_definition*>}

|{<*template_function_definition*><*vhdl_string*>}

<*vhdl_string*>::=<*vhdl_string_no_containing_template_function_definition*>

Note that the argument *arg3* may be missed.This template function returns a value of the string specified by *arg2* if *arg1* is **true** and it returns a value of *arg3* if *arg1* is **false**. When this function has two arguments it returns a value of *arg2* if *arg1* is **true** , otherwise it returns a <null_string>, i.e. a space.

Examples

Let *strob*:=0. Than we will have:

```
@if[strob=1, {: IN BIT_VECTOR (0 TO 7);},{  }]=   .
```

Let *strob*:=1 and *n*:=4 . Than we will have:

```
@if[strob=1,{: IN BIT_VECTOR(0 TO 7);},{ }]=: IN BIT_VECTOR(0 TO 7);
@if[strob=1,{IF(E='0') THEN Y<=@gen[n,{ and }]; ELSE}]=
IF(E='0') THEN Y<= x1 and x2 and x3 and x4; ELSE.
```

Four-argument-functions

<*function_gen*>::=@***gen***[<*arg1*>,<*arg2*>{{,<*arg3*>}}{{,<*arg4*>}}]

<*function_gen*>::=@***gen***[<*arg1*>,<*arg2*>]

<*arg1*>::=<*template_variable*>

<*arg2*>::={<*separator*>}

<*arg3*>::={<*symbol_specified_by_user*>}

<*arg4*>::=<*initial_value*>

<*initial_value*>::=<*natural*>

<*separator*>::= <*vhdl_symbol*>| <*concatenation_of_vhdl_symbols*>

<*concatenation_of_vhdl_symbols*>*::=*<*vhdl_symbol*><*vhdl_symbol*>

| <*concatenation_of_vhdl_symbols*><*vhdl_symbol*>

The argument *arg4* may be missed. In this case it is assumed that its value is equal to 1. With the *arg3* user may specify a symbol, the beginning of each item of the string to be generated (by the item we mean a substring before its concatenation with the separator). Note that if this argument is missed, it is assumed that *x* is the very symbol of a substring. The argument *arg1* specifies a number of substrings to be generated. A length of the *string* defined by the function is expressed with the following formula:

*string*_length = (*symbol*_length+*b*) ∗ *a*+ *separator*_length ∗ (*a* - 1),

where *a* is a value of the first argument of the function, *b* is a length of the value of argument *arg4.*

Examples

```
n:=1; <separator> ::=,              @gen[n,{,}] =x1
n:=2; <separator> ::=,              @gen[n,{,}] =x1,x2
n:=3; <separator> ::=,              @gen[n,{,}] =x1,x2,x3
n:=4; <separator> ::= and           @gen[n,{ and }]=x1 and x2
and x3 and x4.

n:=1; <separator>:=,        @gen[n,{,},{a},1]=a1
n:=2; <separator>:=,        @gen[n,{,},{a},0]=a0,a1
n:=3; <separator>:=,        @gen[n,{,},{a},4]=a4,a5,a6
n:=4; <separator>:=,        @gen[n,{,},{I}]=I1,I2,I3,I4
n:=3; <separator>:= and     @gen[n,{ and },{Y},0]=Y0 and Y1 and
Y2 and Y3
```

<*function_for*>::=

@***for***[<*loop_parameter*>,<*initial_value*>,<*range*>,<*vhdl_text*>]

<*initial_value*>::=<*natural* >|<*template_expression*>

<*loop_parameter*>::=<*template_variable*>

<*range*>::=<*natural*>|<*template_variable*>|<*template_expression*>

<*vhdl_text*>::={<*vhdl_string*>}|{<*vhdl_text*><*template_function_definition*>}

|{<*template_function_definition*><*vhdl_text*>}

This function is applied to specify the repeating fragments of the VHDL text while the text may contain the template functions as well. It is assumed that the loop parameter varies from the initial value specified by <*initial_value*> in the ascending order with the step equal always to 1 until the range of a value defined by <*range*> is achieved. The function may be applied for generating of strings of VHDL text in the case when a value of one string distinguishes from another one only by a value of either the loop parameter or parameters if nested loops are applied . Examples will be given after a definition of <*function_con*>.

Functions with an unspecified number of arguments

<*function_con*>.::=@***con***[<*argument_list*>]

<*argument_list*>::=<*argument*>,<*argument*>|<*argument_list*>,<*argument*>

<*argument*>::={<*vhdl_string*>}|<*template_function_definition*>

|<*template_variable*>

The function performs the concatenation of the argument values in the sequential order from left to right.

Examples

```
@con[{A},{a},{B}]=AaB
Let n:=3 ; r:=4
@con[@gen[n,{,},{Y}],{: OUT BIT; }]=Y1,Y2,Y3: OUT BIT;
 @con[@gen[n,{,},{I}],{:IN BIT_VECTOR(0 TO },
     @sub[2^[r+1]-1], {);}
     ]=I1,I2,I3: IN BIT_VECTOR(0 TO 31);
```

Let *n*:=4 ; *r*:=2;.

```
@for[k,1,n,
   @con[{I},k,{:IN BIT_VECTOR(0 TO },@sub[2^[r+1]–1],
              {);},{ Y},k,{: OUT BIT; }
       ]
    ]=  I1:IN BIT_VECTOR(0 TO 7);Y1: OUT BIT;
        I2:IN BIT_VECTOR(0 TO 7);Y2: OUT BIT;
        I3:IN BIT_VECTOR(0 TO 7);Y3: OUT BIT;
        I4:IN BIT_VECTOR(0 TO 7);Y4: OUT BIT;
```

Let r:=1

```
@for[ k, 0 ,[2^[r+1]–1],
     {WHEN "@b2b[k,r+1]"   Y<=I(@sub[k]) AFTER TOTAL_DELAY; }
    ]
```

This function returns the following value:

```
WHEN "00"   Y   <= I(0) AFTER TOTAL_DELAY;
WHEN "01"   Y   <= I(1) AFTER TOTAL_DELAY;
WHEN "10"   Y   <= I(2) AFTER TOTAL_DELAY;
WHEN "11"   Y   <= I(3) AFTER TOTAL_DELAY;
```

<*function_case*>::=@***case***[<*arg1*>,<*argument_list*>]

<*arg1*>::=<*template_variable*>|<*template_expression*>

<*argument_list*>::=<*argument*>|<*argument_list*>,<*argument*>

<*argument*>::=<*vhdl_text*>

A space of feasible values of the argument *arg1* is defined as follows.

<*template_variable_value*>::=<*natural_excluded _zero*>

<*natural_excluded_zero*>::=<*digit_without_zero*>

|<*natural_excluded_zero*><*natural_digit*>

<*digit_without_zero*>::=1|2|3|4|5|6|7|8|9

Examples

Let *a*:=2. Then

```
@case[a, {BIT_VECTOR(0 TO 15);},{BIT;},{STD_LOGIC;}]=BIT;
```

Let *a*:=3. Then

```
@case[a, {BIT_VECTOR(0 TO 15);},{BIT;},{STD_LOGIC;}]= STD_LOGIC;
```

So far we have introduced several template functions for generating strings of the basic language The power of a function may be enhanced if its argument is another function as it has been shown above. Each function from a given list, i.e. the notation of a function is a *non-terminal* symbol of the *metalanguage*. Metalinguistic formula in BNF notation given above is a description of the syntax of that language. Note that

template variables are *global*, i.e., they are valid in each function argument list of the metaprogram but they are not valid in VHDL text..

Having a formal definition of the template function, now we can define a more complicated structures as follows.

<reuse_component>::=*<reuse_template><user_interface>*

<reuse_template>::=*<vhdl_fragment><template_function>*

|*<template_function><vhdl_fragment>*

|*<reuse_template><vhdl_fragment>*

<user_interface>::=*<argument_values_for_template_variables>*

<metaprogram>::=*<template_function>*|*<metaprogram><template_function>*

<template_function>::=*<function_**sub**>*|*<function_**b2d**>*|*<function_**d2b**>*|

*<function_**if**>*|*<function_**gen**>*|*<function_**fo**r>*|*<function_**con**>*|*<function_**case**>*

<vhdl_fragment>::=*<syntactically_complete_vhdl_text>*

|*<syntactically_incomplete_vhdl_text>*

<syntactically_complete_vhdl_text>::=*<entity_description>*|*<configuration_description>*

|*<architecture_body>*|*<subprogram>*|*<package>*

|*<sequential_statement>*|*<concurrent_statement>*

<syntactically_incomplete_vhdl_text>::=*<part_of_entity_description>*

|*<part_of_configuration_description>*|*<part_of_architecture_body>*|*<part_of_subprogram>*

|*<part_of_package>*|*<part_of_sequential_statement>*|*<part_of_concurrent_statement>*

Template design problem

To describe the template design problem, the following information is accepted as an initial data for a template designer:

a) Characteristics of a class for a given domain object for which the template is to be designed
b) VHDL models and a concrete model for a given representative from the given class;

c) The list of the template functions discussed above (syntactic rules of the functions) .

Initial data a) and b) are introduced to the designer via domain analysis (DA) for reuse[]. Since a template has been formally defined as a program of higher level than the basic program, the template designing procedure is similar to that used in the program development (coding) phase. Several requirements are essential to that procedure:

1) To enhance the reusability of a template, the amount of characteristics and features from the object of a given class and VHDL models respectively as much as possible should be transferred and implemented into the template;
2) To implement those characteristics and features, the template functions should be used.
3) The template implementation procedure must be performed in a proper way. This means not only that the functions should be written according to the syntactical rules but they must be inserted into the basic program in such a manner which will not affect semantics of the code of the basic program produced after pre-processing of the template.
4) Along with the template development the user interface should be built in order to manage the pre-processing procedure properly. As a result a *reusable component* should be built (see formal definition above).
5) As a consequence of the requirement 3, two validation procedures for the correctness of the reusable component are to be performed. Those are pre-processing and modelling.

It is assumed that a reusability of the component can be measured by the number of features and characteristics added to that template of the component.

How reusable component or template is designed

The design procedure is generalised in Figure 1. At the beginning a designer

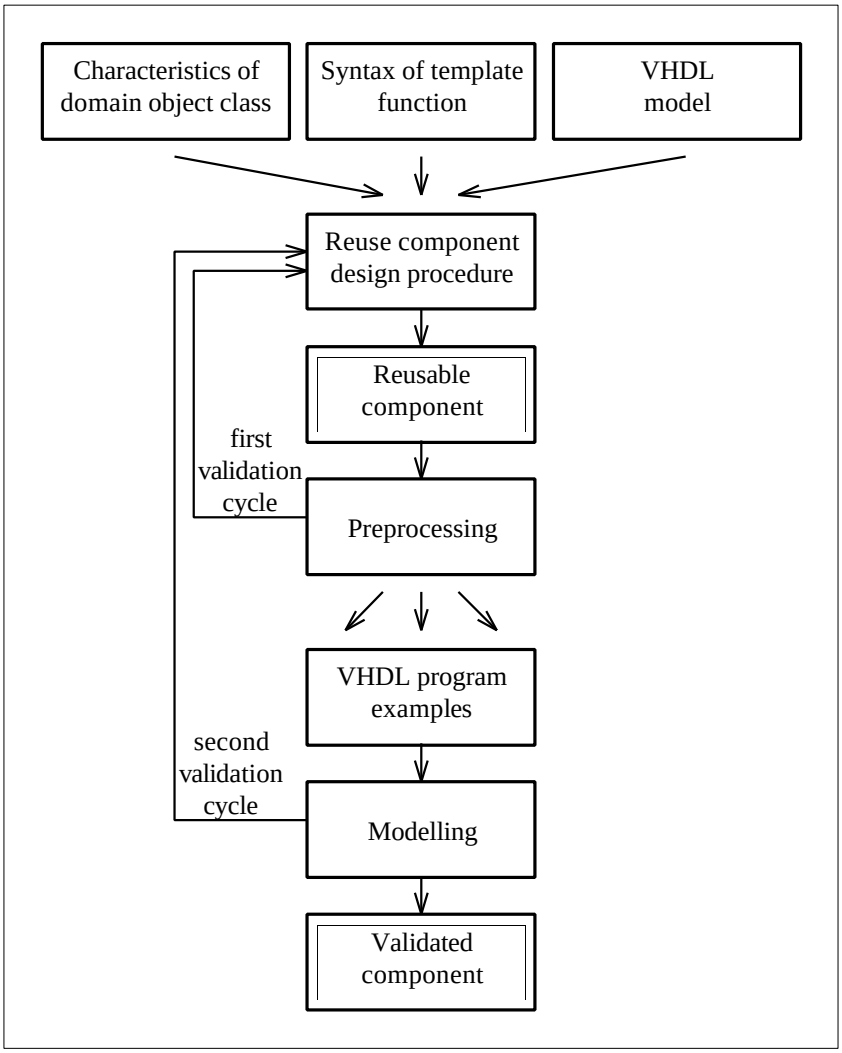

Figure 1. A generalised reusable component design procedure

must have a concrete example of VHDL model that represents a sample of the object from the given class. Since this model is introduced via DA for reuse, it is assumed that its syntax and semantics is already validated. Usually the model implements only one or few features from the VHDL models and a few characteristics only from the set of those which are common to the entire class of the object to be examined. The designer's main task is to add new characteristics and features and this extension is to be done as large as possible. The extension procedure means a *re-coding* of the given VHDL code within the insertion of the appropriate external (template) functions by which new features are to be introduced.

Let us consider an example related with the object class such as MUX (multiplexer). The concrete MUX's VHDL model is shown in Figure 2. This model inherited the following characteristics given for the whole class of the

```
ENTITY MUX2 IS
PORT(  I1 : IN BIT_VECTOR(0 TO 3);
       Y1 : OUT BIT;
       X1, X2: IN BIT);
END MUX2;
ARCHITECTURE MUX2_BEHAVE OF MUX2 IS
BEGIN
     PROCESS (X1, X2)
      VARIABLE S:BIT_VECTOR(0 TO 1);
          BEGIN
          S<= X1&X2;
          CASE S IS
            WHEN "00" Y1 <= I1(0);
            WHEN "01" Y1 <= I1(1);
            WHEN "10" Y1 <= I1(2);
            WHEN "11" Y1 <= I1(3);
          END CASE ;
     END PROCESS;
END MUX2_BEHAVE2;
```

Figure 2. One channel -two address MUX

object: two addresses, one channel, no output delay (or delta delay) (see the slightly shaded terminal nodes on the tree in Fig 3a). This model contains as well one feature inherited from the VHDL models' tree (see Fig 3b).

It is needed to note that the second tree represents the features of the entire domain in the sense of VHDL capabilities[], while the first tree represents only

those characteristics which are essential for the object of a given class.

The outcome of the design is a reusable component. It is shown in Figure 4. This designed model implements these characteristics and features which are represented by the shadowed nodes on the trees (see Figure 3a, 3b). A user

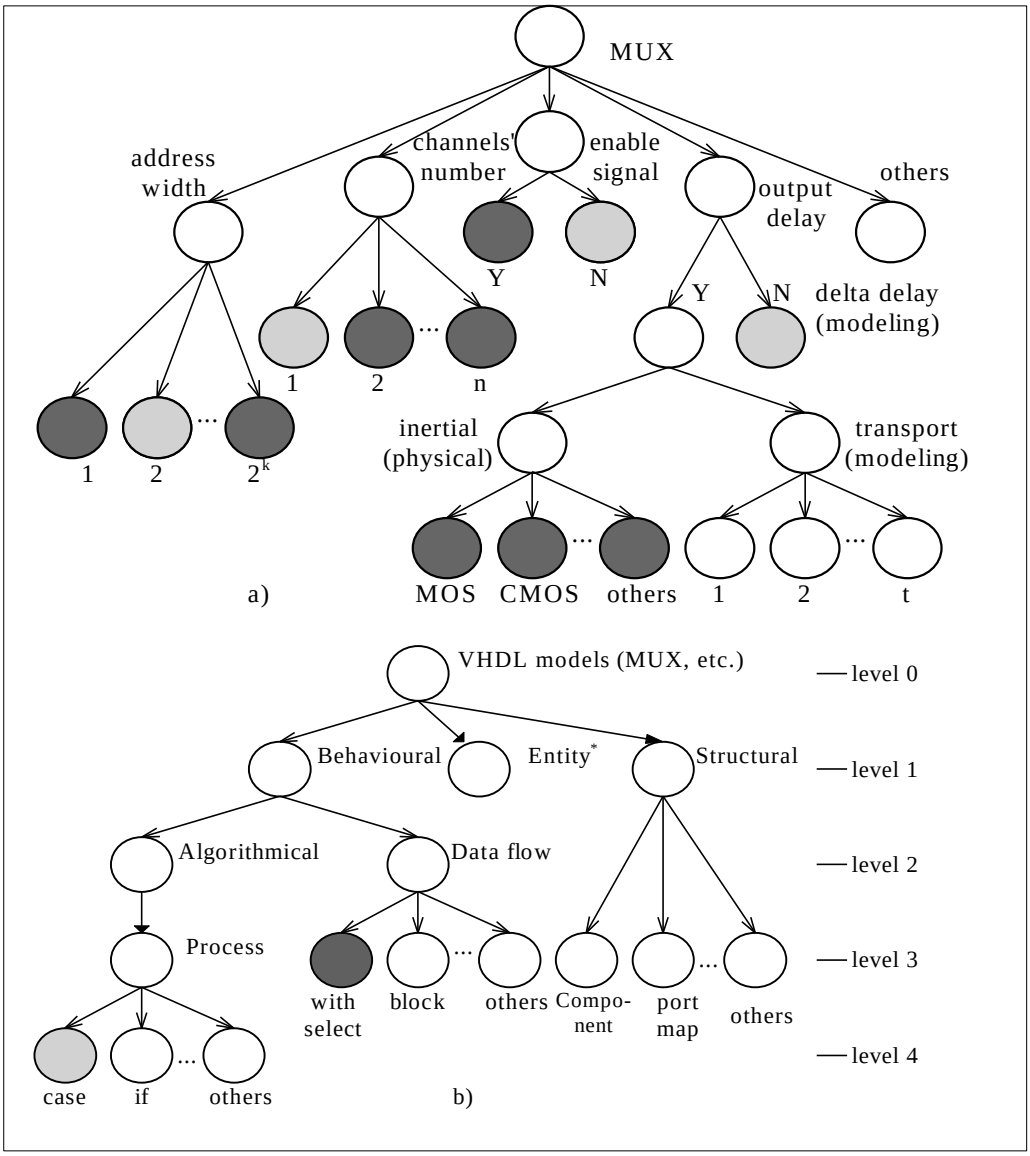

Figure 3. Representation of domain artifacts for the object MUX: (a) characteristics 'tree: (b) VHDL models' tree

interface, the constituent part of that component, is given there as well. It is a very important part of the design because it contains information regarding the semantics of the component used at the instantiated phase via pre-processing.

```
ENTITY MUX@sub[n]_@sub[r] IS
    GENERIC(TOTAL_DEL : TIME := @sub[arg] NS);
    PORT(@for[k,1,n,
         {I@sub[k]:IN BIT_VECTOR(0 TO @sub[2^r–1] );
                    Y@sub[k]: OUT BIT; }
           ]
         @if[enable=1,{E :IN BIT; }]
         @gen[r, {,}]: IN BIT);
END MUX@sub[n]_@sub[r];
ARCHITECTURE MUX_BEHAVE@sub[n]_@sub[r] OF MUX@sub[n]_@sub[r] IS
 BEGIN
 @if[case=0,{ S:=@gen[r,{&}]; WITH @if[enable=1,{E&S},{S}]
SELECT @for[j,1,n, {@for[k,0,2^⌈r+enable⌉–1,
       { Y@sub[j]<=@if[k <[2^r]*enable,{'0'},
                              {I@sub[j](@sub[k-[2^r]*enable])}
                        ]AFTER TOTAL_DELAY WHEN "@d2b[k,r+enable] }
                          ]
                    }
           ]
             },
  {PROCESS (@gen[r,{,}] @if[enable=1,{,E}])
  VRIABLE S: BIT_VECTOR(0 TO @sub[r-1]);
  BEGIN
    S:= @gen[r, {&}];
@if[enable=1, {IF (E = '0') THEN
       @for[k,1,n, @con[{Y},k,{<= '0';}]]
      ELSE    }
   ]
  @for[j,1,n,       {CASE S IS
          @for[k,0, [2^r–1],
            {WHEN "@d2b[k,r]" =>Y@sub[j]<=I@sub[j](@sub[k])
                  AFTER TOTAL_DEL;
            }
             ]
          END CASE;
                    }
       ]
 @if[enable=1, { END IF; }]
      END PROCESS;
 }
    ]
END MUX_BEHAVE@sub[n]_@sub[r];
```

user interface

promts for user and *user's answers* which may change default values	space of feasible values	default values
Specify output delay expressed with ns? *10*	{**0**,1,2,3…}	*arg:*=**0**
Enter MUX's channels number? *2*	{**1**,2,3,…}	*n:*=**1**
Enter MUX address width ?	{1,**2**,3,…}	*r:*=**2**
Does MUX have enable signal ? *1*	{**0**,1}	*enable:*=**0**
Is MUX given by CASE or WITH construct?*1*	{**0**,1}	*case:* =**0**

Figure 4. Reuse template and user interface for multiplexer

The interface defines the space of allowable values of arguments to be transferred for the template functions. Arguments or, in other words, template variables have default values which may be changed by a user.

If this is done as specified by the interface, the following VHDL program will be produced automatically (see Figure 5).

```
ENTITY MUX2_2 IS
   GENERIC (TOTAL_DEL : TIME := 10 NS);
   PORT(  I1 : IN BIT_VECTOR(0 TO 3);Y1 : OUT BIT;
          I2 : IN BIT_VECTOR(0 TO 3);Y2 : OUT BIT;
             E :IN BIT;
             X1, X2: IN BIT);
END MUX2_2;

ARCHITECTURE MUX_BEHAVE2_2 OF MUX2_2 IS
BEGIN
       PROCESS (X1, X2,E)
        VARIABLE S: BIT_VECTOR(0 TO 1);
         BEGIN
         S:= X1&X2;
           IF (E='0') THEN
                      Y1<= '0';
                      Y2<= '0';
                      ELSE
              CASE S IS
                WHEN "00" Y1 <= I1(0) AFTER TOTAL_DEL ;
                WHEN "01" Y1 <= I1(1) AFTER TOTAL_DEL ;
                WHEN "10" Y1 <= I1(2) AFTER TOTAL_DEL ;
                WHEN "11" Y1 <= I1(3) AFTER TOTAL_DEL ;
              END CASE ;
              CASE S IS
                WHEN "00" Y2 <= I2(0) AFTER TOTAL_DEL ;
                WHEN "01" Y2 <= I2(1) AFTER TOTAL_DEL ;
                WHEN "10" Y2 <= I2(2) AFTER TOTAL_DEL ;
                WHEN "11" Y2 <= I2(3) AFTER TOTAL_DEL ;
              END CASE ;
        END IF;
       END PROCESS;
END MUX_BEHAVE2_2;
```

Figure 5. VHDL MUX model generated with respect to the user defined interface

Discussion and concluding remarks

Hardware modules are highly reusable structures, and VHDL models which describe them are reusable, too. However, the flexibility and the scope of reusability of those models can be enhanced if external reuse mechanism are applied. Using external functions we can add new features and characteristics to those which already exist in a given VHDL model. A VHDL model enriched by the external functions to introduce new features and characteristics along with the users' interface we call a reusable component. User interface aims to transfer argument values for the external functions

when the instantiation process via pre-processing is performed. Furthermore, through the user interface we can determine the space of feasible values of those arguments and manage the pre-processing procedure. The scope of reusability and flexibility we may measure by the number of characteristics and features that reusable component implements. For example, in the case of the component of MUX discussed above, we have the total number of characteristics as follows: *n*r*t*l.* If for example, *n*=4, *r*=6, *t*=3 (delta delay, inertial delay, transport delay), *l*=2 (**case** and **with select** constructs), this item is expressed by 144. In order to receive correct MUX model in VHDL (one of 144) automatically, only a few figures are needed to change by a user. Since our reusable component implements the different language constructs, the model is adjustable to the requirements of different tools for modeling and synthesis (it is a well-known problem that not each VHDL construct is synthesisable now).

The introduced functions, as they were defined in this context, are universal vehicle in the following sense:

1) since an argument of a function may be another function, they support the hierarchy of the basic language and can be inserted into the text of the basic program practically at any place without restrictions;
2) they allow to perform any transformations with a text of the basic language (not only to change or substitute the constants as generic do);
3) functions are the language undependable mechanism and might be applied to compose software reusable components in other high level languages, too. In this case only one restriction must be considered - symbols by which we define the functions must be different from those used in a basic language.
4) A component is defined as a generic construct consisting of a VHDL template with the incorporated functions and interface. This reduces the needed number of components. However, the interface is a component-dependable.

References

[1] Agsteiner, K., D. Monjau, S. Schulze (1995). Object-Oriented High-Level Modeling of System Components for the Generation of VHDL Code. European Design Automation Conference, pp. 436-441.

[2] Ashenden, P. J. (1995). The Designer's Guide to VHDL. Morgan Kaufmann Publisher, Inc., 688 pp.

[3] Börger, E., U. Glässer, W. Müller (1994). The Semantics of Behavioral VHDL'93 Descriptions. European Design Automation Conference, pp. 500-505.

[4] Bassett, P. G. (1997). The Theory and Practice of Adaptive Reuse. SSR'97, MA, USA, pp. 2-9

[5] Frakes, W. B., Ch. J. Fox (1996). Quality Improvement Using a Software Reuse Failure Modes Model. IEEE Transactions on Software Engineering, Vol. 22, No. 4, pp. 274-279.

[6] Gall, H., M. Jazayeri, R. Klösch (1995). Research Directions ir Software Reuse: Where to go from here?, SSR'95 , Seattle, WA ,pp. 225-228.

[7] Henninger, S. (1995). Developing Domain Knowledge Through the Reuse of Project Experiences. European Design Automation Conference, pp. 186-195.

[8] Kission, P., H. Ding, A. A. Jerraya (1995). VHDL Based Design Methodology for Hierarchy and Component Re-Use. European Design Automation Conference, pp. 470-475.

[9] Narayan, S., D. D. Gajski (1993). Features Supporting System-Level Specification in HDLs. European Design Automation Conference, pp. 540-545.

[10] Poulin, J. S. (1995). Populating Software Repositories: Incentives and Domain-Specific Software. J. Systems Software, No. 30, pp. 187-199.

[11] Preis, V., R. Henftling, M. Schütz, S. März-Rössel (1995). A Reuse Scenario for the VHDL-Based Hardware Design Flow. European Design Automation Conference, pp. 464-469.

[12] Rajlich, V., J. H. Silva (1996). Evolution and Reuse of Orthogonal Architecture. IEEE Transactions on Software Engineering, Vol. 22, No. 2, pp. 153-157.

[13] Symposium on Software Reusability (SSR'95, 96, 97).

[14] Tracz, W. (1990) Software Reuse: Emerging Technology (tutorial). IEEE Computer Society Press, 378p.

[15] Vahid, F., S. Narayan, D. D. Gajski (1994). A Transformation for Integrating VHDL Behavioral Specification with Synthesis and Software Generation. European Design Automation Conference, pp. 552-557.

Received , 1998

V. Stuikys received Ph.D. degree from Kaunas Politechnic Institute in 1970. He is currently the associate professor at Computer Department, Kaunas University of Technology, Lithuania. His research interests include program generation for domain - specific systems, high level domain - specific languages, expert systems, digital signal processing and CAD Systems.

PAKARTOTINOS VARTOSENOS VHDL KOMPONENTØ PROJEKTAVIMAS, PAGRÁSTAS IÐORINËMIS FUNKCIJOMIS

Vytautas Ðtuikys