INFORMATICA, 1998, Vol. 9, No. 4, 491–506
© 1998 *Institute of Mathematics and Informatics, Vilnius*

# Design of Reusable VHDL Component Using External Functions

Vytautas ŠTUIKYS
*Kaunas University of Technology*
*Studentų 50, 3031 Kaunas, Lithuania*
*e-mail: vytas.stuikys@if.ktu.lt*

Received: February 1998

**Abstract.** This paper describes a method how to represent and build a reusable VHDL component. By that component we can, for example, describe a family of the relative VHDL models. To represent the component, we use external functions as a mechanism to support a pre-processing and perform the instantiation of the component. A user interface, the constituent of the reusable component, serves for transferring parameters for the instantiation. We deliver a formal syntax of the functions and examples of their semantics. We describe the design of the reusable component as a procedure of transferring of: a) the intrinsic characteristics for a given family of domain objects and b) features from a given VHDL model(s). Those features require to be re-coded and extended with new ones by means of the external functions introduced. To test a reusable component, we use pre-processing and modelling.

**Key words:** reusable component, template, VHDL model, pre-processing, external functions.

## 1. Introduction

Our aim is to work out a framework for the development of a reusable VHDL component, the building block for a virtual library. Reusability is a very wide and broadly discussed topic. The treatment of various aspects of the reusability problems is in numerous publications on Software Engineering (Frakes and Fox, 1996; Gall *et al.*, 1995; Karhinen *et al.*, 1997; Kission *et al.*, 1995; Lenz *et al.*, 1987; Poulin, 1995; Tracz, 1990) and hardware designs based on VHDL models (Agsteiner *et al.*, 1995; Börger *et al.*, 1994; Kission *et al.*, 1995; Henninger, 1995; Narayan and Gajski, 1993; Preis *et al.*, 1995; Rajlich and Silva, 1996; Vahid *et al.*, 1994), too. In component-based reuse (Biggerstaff and Richter, 1990; Lenz *et al.*, 1987; Tracz, 1990) at least two problems are essential. The first problem is how to represent components (knowledge, artefacts or assets) (Biggerstaff and Richter, 1990; Gall *et al.*, 1995). The second one is how to modify easily the available components (Agresti and McGarry, 1990; Mili *et al.*, 1997). The analysed representation models vary in a wide range from the verbatim description, code templates to knowledge-based, frame-based and frameworks models (Agresti and McGarry, 1990; Bassett, 1987; Henninger, 1995; Johnson, 1997). Among other known mechanisms to bind the components' representation with the ability to modify them easily are macro extension and pre-processing. Compiler flags are the main mechanism to represent a component for a pre-processor. Such a method, for example, is analysed for configuring software systems (Karhinen *et al.*, 1997). The paper (Bassett, 1997) describes a more complicated mechanism based on the pre-processing commands use.

Our approach is similar to the above mentioned one. Instead of commands, however, we use external functions. The returned value of a function is a string of the VHDL text which, in turn, may contain other functions. The nested functions allow us to achieve a deep and flexible external modification of the VHDL code to be pre-processed. Furthermore, we introduce a specific function, the *generate* function, that produces the available string for insertion and modification of the given code. We believe that the proposed method can significantly extend the internal reuse possibilities (packages, generics, etc.) of the standard language (Ashenden, 1995).

We use the following terminology. A pre-processing text, the VHDL code enriched by the external functions, we call a *reusable template. User's interface* is a structure which represents a space of allowable values of parameters and the specific values of the parameters to be transferred to the external functions' arguments for their instantiation. A *reusable component* is a *reusable template* plus *user's interface*.

The main task of the discussion is to show how the pre-processing mechanism based on the external functions works in detail. In Section 2 we introduce a formal syntax of the external functions and examples of their semantics. In the Sections 3 and 4 we describe a procedure of the reusable component design. In Section 5 we present a discussion and concluding remarks.

## 2. A Formal Syntax of External Functions

We will introduce a list of functions called external or *template functions*. Each function has a list of arguments and 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 VHDL. 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 or template expression 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>*::= ***substitute*** | ***binary_to_decimal*** | ***decimal_to_binary***
| ***generate*** | ***branch*** | ***for_loop***| ***concatenate*** | ***case***

Note that the symbol "]", the end of the argument list, indicates also the end of the function. We consider the functions in the order of a number of arguments they have. So, we discuss first the functions with one argument. The argument list *<argument_list>* (or in the simplest case *<arg>*) will be separately defined for each function.

**One-argument-functions**

*<function_sub>*::= @***sub***[*<arg>*]
*<arg>*::=*<template_constant>* | *<template_variable>* | *<template_expression>*
*<template_constant>*::=*<natural>* | *<vhdl_string>*
*<template_variable>*::=*<letter>* | *<template_variable><letter>*
| *< template_variable><digit>*
*<natural>::=<digit>*|*<natural>< digit>*
*<natural_digit>*::=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
*<letter>*::=a | b | c |... | z
*<vhdl_string>*::=*<vhdl_symbol>*| *<concatenation_of_vhdl_symbols>*
*<concatenation_of_vhdl_symbols>*::=*<vhdl_string><vhdl_string>*|
*<concatenation_of_vhdl_symbols><vhdl_string>*
*<template_expression>*::=*<factor><arithmetic_operator><factor>*
|[*<template_expression>*]
|*<template_expression><arithmetic_operator><factor>*
*<factor>*::=*<natural>*|*<template_ variable>*
*<arithmetic_operator>*:: = $+| - | * |/|\hat{}\,|$ *mod*

For example, let $k = 5$; then @***sub***$[k] = 5$ @***sub***$[2\hat{}k] = 32$, @***sub***$[2\hat{}k + 2 * [k + 1]]$ $= 44$.

*<function_b2d>*::=@***b2d***[*<binary_of_natural>*]
*<binary_digit>*::=0|1
*<binary_of_natural>*::=*<binary_digit>* |*<binary_of_natural><binary_digit>*

For example, @***b2d***$[101] = 5$.
*<comment>*::=@ – *<string_of_text>*
For example, @ – this is a string of comment, i.e., any available sequence of symbols.

**Two-argument-function**

*<function_d2b>*::=@***d2b*** [*<list_item>,<list_item>*]
*<list_item>*::=*<template_variable>* | *<template_expression>*
The first argument means a 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>::=<logic_expression_item><relation_operator>*
*<logic_expression_item>*
*<logic_expression_item>::=*
*<natural>* | *<template_variable>* | *<template_expression>*
*<relation_operator>*::= < | > | = | <= | >= | /=
*<arg2>::=<text>*
*<arg3>::=<text>*
*<text>::={<vhdl_string>}*| *<template_function_definition>* |{*<extended_text>*}
*<extended_text>::=<template_function_definition><vhdl_string>*
|*<vhdl_string><template_function_definition>*
|*<extended_text><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. An argument $arg2$ or $arg3$ (if it is not skipped) may contain a VHDL text which, in turn, may contain a template function. The nested functions allow to a designer to carry out a flexible modification of the text to be pre-processed (see the third example below).

*Examples*

Let $strob = 0$. Then we will have:
@***if*** [$strob = 1$, {: IN BIT_VECTOR (0 TO 7); }, { }] = .
Let $strob = 1$ and $n = 4$. Then 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>* | *< template_expression>*
*<arg2>::={<separator>}*
*<arg3>::={<vhdl_symbol>}|{<vhdl_string>}*
*<arg4>::=<initial_value>*
*<initial_value>::=<natural>* | *< template_expression>*

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

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 the argument $arg3$ is skipped, *x* is assumed as a symbol of that 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>*,*<text>*]
*<initial_value>*::=*<natural>*| *<template_expression>*
*<loop_parameter>*::=*<template_variable>*| *<template_expression>*
*<range>*::=*<natural>*|*<template_variable>*| *<template_expression>*
*<text>*::={*<vhdl_string>*}|{*<extended_text>*}| *<template_function_definition>*

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 the 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>*::=*<template_variable>* | *<template_function_definition>* |
{*<vhdl_string>*}|{*<text>*}

The function performs a 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,l,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>*::=*< text>*

A space of feasible values of the argument $arg1$ is defined by a natural number excluding zero.

*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. It allows to exclude the changes in the VHDL text.

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

```
<reuse_component>::=<reuse_template><user_interface>
<reuse_template>::=<vhdl_fragment><template_function_definition>
                  |<template_function_definition><vhdl_fragment>
                  |<reuse_template><vhdl_fragment>
<user_interface>::=<argument_values_for_template_variables>
<metaprogram>::=<template_function_definition>
                  |<metaprogram><template_function>
<template_function_definition>::=<function_sub> | <function_b2d>
                                  | <function_d2b> | <function_if>
                                  | <function_gen> | <function_for>
                                  | <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>
```

## 3. Template Design Problem

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

a) characteristics of a family of the given domain objects (for example, registers, multiplexers, etc.) for which the template is to be designed;

b) VHDL models and a concrete model for a given representative from the given objects' family;

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

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 (features) from the domain objects' and VHDL models 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.

## 4. How Reusable Component or Template is Designed

The design procedure is generalised in Fig. 1. At the beginning a designer must have a concrete example of VHDL model that represents a sample of the object from the given class. Since this model we introduce with the reusability concept in mind (for example, from a functioning system), we assume 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 family 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 *recoding* of the given VHDL code within the insertion of the appropriate external (template) functions by which new features are to be introduced.

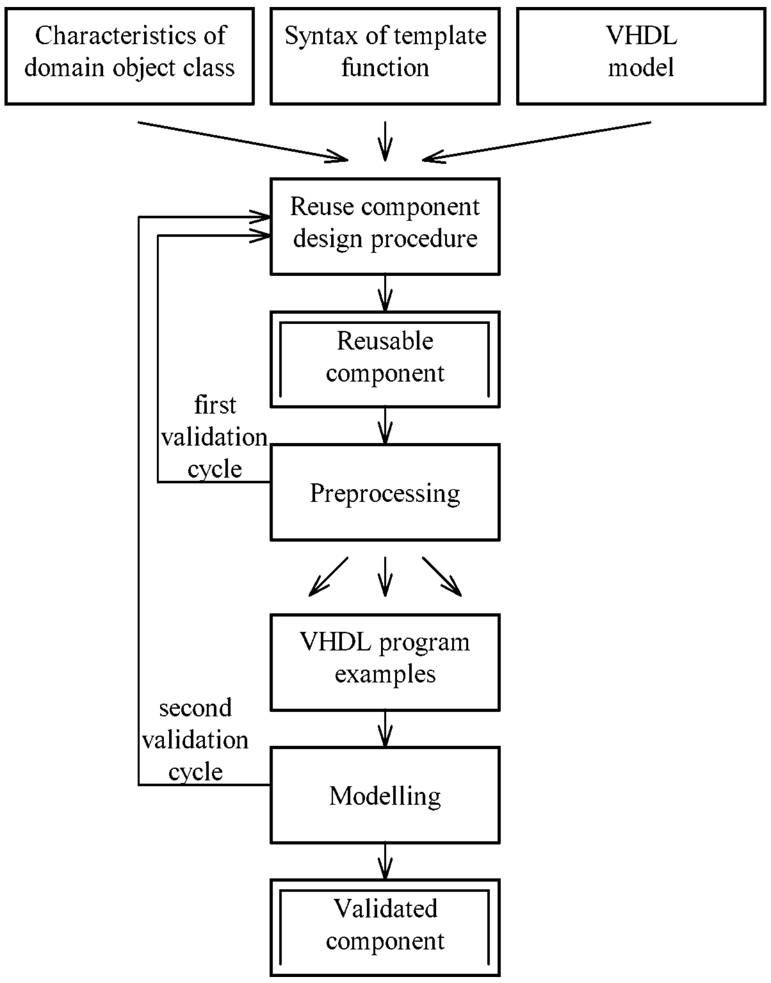

Fig. 1. A generalised reusable component design procedure.

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

Note that the second tree represents the features of the entire domain in the sense of VHDL capabilities (Ashenden, 1995), while the first tree represents only those characteristics which are essential for the object of a given family.

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

```
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;
```

Fig. 2. One channel-two address MUX.

of that component, is given there as well (see Fig. 4b). It is a very important part of the design because it contains information regarding the semantics of the component used at the instantiation phase via pre-processing.

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. The pair of symbols '$' are flags for pre-processor.

If this is done as specified by the interface, the following VHDL program will be produced automatically by a pre-processor (see Fig. 5).

## 5. Discussion and Concluding Remarks

Hardware modules are reusable structures, and VHDL models that describe them are reusable, too. However, the flexibility and the scope of reusability of those models can be significantly enhanced with an *external* reuse mechanism. Using external functions, for example, 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 for introducing new features and characteristics along with the users' interface we call a reusable component. User interface aims to transfer argument (parameter) values for the external functions for the instantiation of that component via pre-processing. Furthermore,

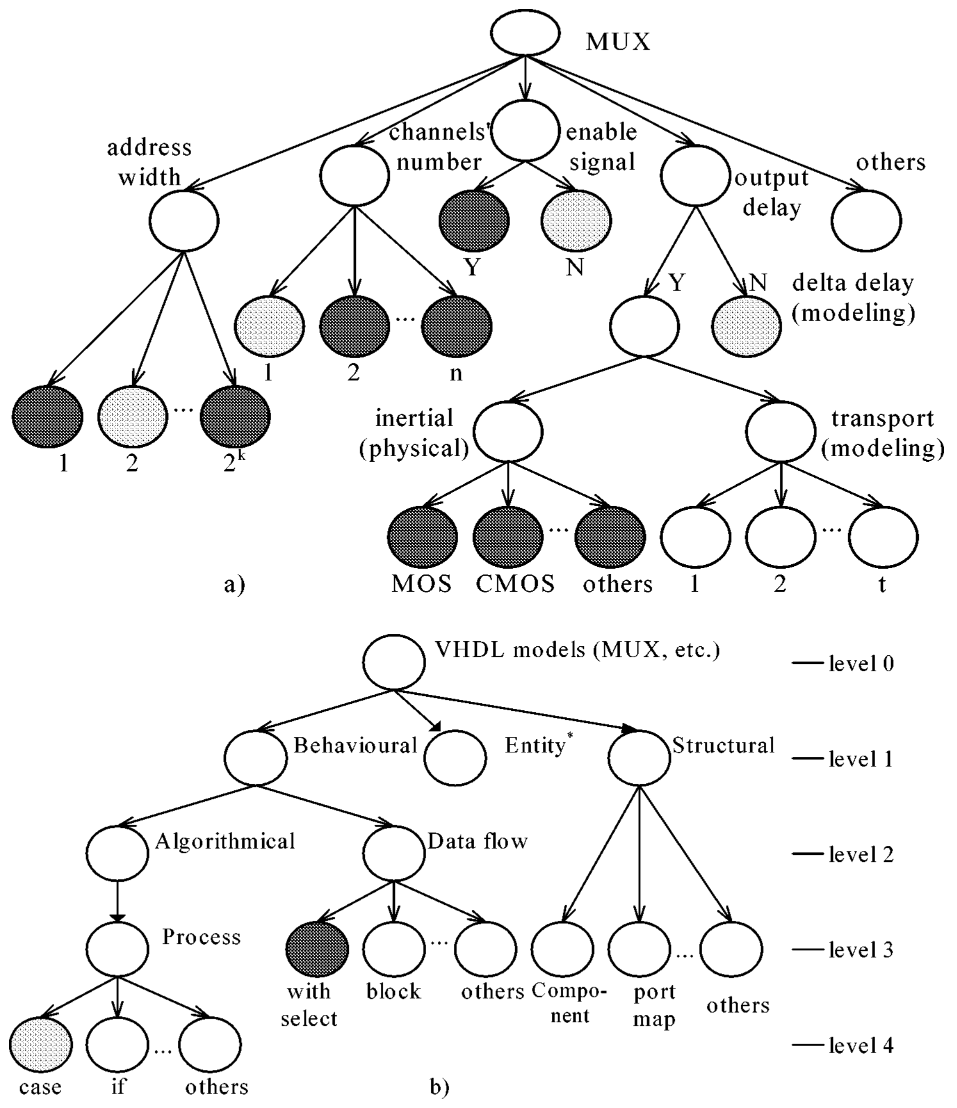

Fig. 3. Representation of domain artefacts for the object MUX: (a) characteristics' tree: (b) VHDL models' tree.

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: $n*r*t*l$. For example, if values of $n$, $r$, $t$, $l$ would be as follows $n = 4$, $r = 6$, $t = 3$ (delta delay, inertial delay, transport delay), $l = 2$ (*case* and *with select* constructs), this number would be 144. In order to receive the correct MUX model in VHDL (one of 144) automatically, user needs to change only a

```
$
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]";}
                         ] @ - end internal "for"
                    }
              ] @ - end external "for"
              },
 {PROCESS (@gen[r, {,}] @if[enable=1,{,E}])
 VARIABLE S: BIT_VECTOR(0 TO @sub[r − 1]);
 BEGIN
   S:= @gen[r, {&}];
@if[enable=1, { IF (E ='0') THEN
        @for[k,1,n, { Y @con[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 internal "for"
          END CASE;
                }
      ] @ - end external "for"
 @if[enable=1, { END IF; }]
       END PROCESS;
 }
    ] @ - end "if" that follows after BEGIN
END MUX_BEHAVE @sub[n]_@sub[r];
$
```

Fig. 4a. Reusable template for multiplexer.

| 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**; |

Fig. 4b. User interface for multiplexer.

```
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;
```

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

few figures. 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 synthesizable 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;
2) Functions are the language-independent mechanism and might be applied to compose software reusable components in other high level languages, too. In this case only one restriction should be considered - symbols by which we define the functions must be different from those used in a basic language;
3) We define a component as a generic construct consisting of a VHDL template with the incorporated functions for the extension of the model applicability and interface. By pre-processing of reusable component, we can create only those VHDL models that are needed at the moment for a given system. This reduces a number of models needed to save.

The nested functions increase the flexibility of modifications. However, they reduce readability and understandability of the component. We suggest to overcome this disadvantage by packaging a reusable component together with the pre-processor that processes it.

**Acknowledgement**

Author would like to thank to anonymous reviewer who's notes have served for an improvement of the paper.

## References

Agsteiner, K., D. Monjau, S. Schulze (1995). Object-oriented high-level modeling of system components for the generation of VHDL code. In *European Design Automation Conference, IEEE*. pp. 436–441.

Agresti, W. W., and F.E. McGarry (1990). The Minnowbrook workshop on software reuse: A summary report. In *Software Reuse: Emerging Technology*. IEEE Computer Society Press. pp. 3–11.

Ashenden, P. J. (1995). *The Designer's Guide to VHDL.* Morgan Kaufmann Publisher, Inc.

Biggerstaff, T., and C. Richter (1990). Reusability framework, assesssment, and directions. In *Software Reuse: Emerging Technology.* IEEE Computer Society Press, pp. 3–11.

Börger, E., U. Glässer, W. Müller (1994). The semantics of behavioral VHDL'93 descriptions. In *European Design Automation Conference, ACM*. pp. 500–505.

Bassett, P.G. (1987). Frame-based software engineering. IEEE Software, 9–16.

Bassett, P. G. (1997). *The Theory and Practice of Adaptive Reuse.* SSR'97, ACM, MA, USA. pp. 2–9

Frakes, W. B., and Ch. J. Fox (1996). Quality improvement using a software reuse failure modes model. *IEEE Transactions on Software Engineering*, **22**(4), 274–279.

Gall, H., M. Jazayeri, R. Klösch (1995). Research directions and software reuse: where to go from here? In *Symposium on Software Reuse'95*. Seattle, WA, pp. 225–228.

Henninger, S. (1995). Developing domain knowledge through the reuse of project experiences. In *European Design Automation Conference, IEEE*. pp. 186–195.

Johnson, R.E.(1997). Components, frameworks, patterns (extended abstract). In *Symposium on Software Reuse'97*. ACM, MA, USA. pp. 10–17.

Karhinen, A., A. Ran., T. Tallgren (1997). Configuring designs for reuse. *Symposium on Software Reuse'97*. ACM, MA, USA. pp. 199–208.

Kission, P., H. Ding, A. A. Jerraya (1995). VHDL based design methodology for hierarchy and component re-use. In *European Design Automation Conference, IEEE*. pp. 470–475.

Lenz, M., H.A. Shmit, P.W. Wolf (1987). Software reuse from building bloks. *IEEE Software*. 34–42.

Narayan, S., D. D. Gajski (1993). Features supporting system-level specification in HDLs. In *European Design Automation Conference, IEEE*. pp. 540–545.

Poulin, J. S. (1995). Populating software repositories: incentives and domain-specific software. *J. Systems Software*, **30**, 187–199.

Mili, R., J. Desharnais, M. Frappier, A. Mili (1997). A calculus of program modifications. *Symposium on Software Reuse'97*. ACM, MA, USA. pp. 157–168.

Preis, V., R. Henftling, M. Schütz, S. März-Rössel (1995). A reuse scenario for the VHDL-based hardware design flow. In *European Design Automation Conference, IEEE*. pp. 464–469.

Rajlich, V., J. H. Silva (1996). Evolution and reuse of orthogonal architecture. *IEEE Transactions on Software Engineering*, **22**(2), 153–157.

Tracz, W. (1990) *Software Reuse: Emerging Technology (tutorial)*. IEEE Computer Society Press.

Vahid, F., S. Narayan, D.D. Gajski (1994). A transformation for integrating VHDL behavioral specification with synthesis and software generation. In *European Design Automation Conference, ACM*. pp. 552–557.

**V. Štuikys** received Ph.D. degree from Kaunas Politechnic Institute in 1970. He is currently an 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.

## Atsikartojančios vartosenos VHDL komponento projektavimas, pagrįstas išorinėmis funkcijomis

Vytautas ŠTUIKYS

Straipsnyje pateikiamas metodas, kaip atvaizduoti ir sukurti atsikartojančios vartosenos VHDL komponentą. Šitoks komponentas sukuriamas: 1) vartojant išorines funkcijas, kuriomis yra aprašomos srities objektų duotos klasės esminės charakteristikos, 2) duotąjį VHDL modelį papildant naujomis savybėmis. Yra pateikta formali funkcijų sintaksė. Komponentas konkretizuojamas per vartotojo interfeisą, kai aktyvizuojamas preprocesorius ir funkcijų argumentai susiejami su konkrečiomis parametrų reikšmėmis.