INFORMATICA, 2000, Vol. 11, No. 1, 71–86

Scripting Language Open PROMOL and its Processor

Vytautas ŠTUIKYS, Robertas DAMAŠEVIČIUS
Kaunas University of Technology
Studentų 50, 3031 Kaunas, Lithuania
e-mail: vystu@if.ktu.lt, damarobe@soften.ktu.lt

Received: December 1999

Abstract. We present the capabilities of the scripting language Open PROMOL and its processor. The intention of the language is to pre-program specifications for modifying programs written in a target language. We use its processor either as a tool for developing the stand-alone reusable components or as a "component-from-the-shelf" in generative tools for generating domain specific programs. The processor itself uses the module (lexical analyser and parser) produced by Lex & Yacc as a reusable component. We describe the generation, computation, control, parameterization and gluing capabilities of the language. We compare our approach with the similar approaches known in the literature.

Key words: scripting language, target language, program modification, component-based reuse, generative reuse, VHDL, domain specific program.

1. Introduction

The very nature of any program requires the predefined changes, modifications or transformations. A program can be or have to be modified in two different ways: at *its running time phase* or at *its construction phase*. We consider the program modification at its construction phase.

A very important reason why we need to modify a program is the need to increase its reusability adapting the program to the different contexts of its use. Program reuse or more generally, software reuse, is widely believed to be a key in achieving higher productivity, better quality and reliability. Software reuse rely on the use of well-established reusable components and composition of a software system from those components.

The composition process, as well, can be implemented in the tools that generate a software system. *Software system generators* are the foremost achievements in software reuse. Both, the stand-alone reusable components and the software system generators, require *sophisticated modifications and adaptations* of the components read from the reuse libraries.

The generally accepted way to produce a reusable component is to apply some *abstraction* and *generalisation* at the component construction phase. We argue in this paper that the component generalisation can be achieved at its construction phase through the

pre-programmed modifications and *adaptations* of its instance given in the target language.

The natural way to express modifications is to write *a specific external program*, other (higher) than a target program that we need to modify. But this external program can't be dealt with separately from the given instance to be modified. To support the implementation of this concept, we need to have a *specific language* (sometimes referred to as *scripting, glue, system integration or composition* language (Ousterhout, 1998)) for specifying the target program modifications. Both languages, the scripting language and the target language, are complementary.

In this paper we suggest *the scripting language* called **Open PROMOL** (**PRO**gram **MO**dification **L**anguage) and its *processor* for the development of stand-alone reusable components described in *any domain-specific language*, as well as for incorporating it into domain-specific program generators.

The structure of the paper is as follows. In Section 2 we analyse the related works. In Section 3 we deliver the main concept and basic features of the language with the illustrative examples to demonstrate its modification capabilities. In Section 4 we present the Open PROMOL processor. In Section 5 we present the extended examples to demonstrate gluing capabilities for building of monolithic components (internal gluing), as well as structural composition of independent separate components (external gluing). In Section 6 a reader can find the description of preliminary experiments executed in different domains and carried out with the Open PROMOL Processor. In Section 7 we evaluate the achieved results. Finally, in Section 8 we formulate concluding remarks and problems for further consideration.

2. Analysis of the Related Works

In recent years the component-based software engineering attracted much attention from researchers (Sametinger, 1997; Kozaczynski *et al.*, 1998; Brown *et al*., 1998; Broy *et al*., 1998; Bassett, 1999) as a separate direction in software reuse, as well as a generative approach (Batory *et al*., 1994; 1995; 1997; 1998; Terry, 1997; Czarnecki *et al*., 1999). Many different viewpoints exist of what software reuse is (Bassett, 1997; Jacobson *et al*., 1997; Sametinger, 1997). Cooper, for example, defines software reuse as "the capability of a previously developed software component to be used again, in part or in its entirety, with or without modifications" (Sametinger, 1997).

Many different approaches have been proposed and analysed in the various contexts for the program modification. In (Pfleeger, 1998), for example, the nature of the software system changes are dealt with in the context of its life cycle. A modification based on formal calculus has been introduced in (Mili *et al*., 1997). Bassett distinguishes very clearly the program modifications at its construction phase from the run time modifications (Bassett, 1997). He has suggested the adaptive reuse concept based on the model that composes a module from the hierarchically adapted *frame-components* using external commands. These commands control parameters for the modification of an external text.

Papers (Batory *et al*., 1994; 1995) describe another model, called P++, that can be treated as a C++ extension for component generalisation with the intention to modify it at the application generation. Other author suggests the pre-processing commands to extend C++ in order to enhance the modification capabilities of the language (Arney, 1998). Authors (Štuikys, 1998; Štuikys *et al*., 1998) apply the external function concept for describing modification of a target program in VHDL.

The papers (Ousterhout, 1993; 1998) drew attention to TCL (Tool Command Language). The author categorises that language as a scripting language for composing components. He defines the roles of the scripting language and the target language with which program is to be modified. A target language serves for the development of components. A scripting language serves for gluing components into system (Schneider, *et al.*, 1999).

More specifically, the target language might be a domain specific language (DSL). Such languages have a long history of own evolution that starts from the manufacturing applications (Rembold *et al.,* 1986). In recent publications (Batory *et al.*, 1998; Hudak, 1998) authors analyse DSL as tools for the development of reusable components. The increased attention to DSL can be explained by two reasons. The first reason is that reuse is much easier to implement for the narrow well-defined domain. The second reason is the less cost at the latest life cycle phase of a software system implemented with DSL (Hudak, 1998).

We use VHDL (Chang, 1997; IEEE Std. 1076, 1996) as a target language for the case study. Our research interests combine both the generative and component-based reuse for the domain-specific applications including VHDL application domains. VHDL (**V**ery High Speed Integrated Circuits (VHSIC) **H**ardware **D**escription **L**anguage) is excellent for the reuse demonstration. The language supports gluing capabilities (a configuration, composition of components, processes, packages and libraries). This allows us better evaluate the gluing capabilities of the PROMOL which is the target language-independent.

3. The Basic Characteristics of the Open PROMOL

Open PROMOL is a representative of the scripting languages (ScL). Its main intention is to specify program modifications of a target language (TL). The **PROMOL** syntax is based on external functions formally described in (Štuikys, 1998). After half-a-year experiments with the processor (Štuikys *et al*., 1998), we have made serious improvements in the previous work. Those include the new functions, more powerful computation capabilities, revised interface, etc. But the main concept of the language has been preserved because *external functions* allow to perform a composition for modifications in a simple and natural way. They always return strings of the TL that are to be composed in some pre-programmed way. The list of the function names is: ***sub***, ***b2d***, ***d2b***, ***base***, ***move***, ***include***, ***change***, ***if***, ***gen***, ***for***, ***case***, ***macro***, ***prompt***. The underlined functions are the newly introduced ones. The list is still open for extensions (that explains why we use the

word *open* before **PROMOL**). The scripting program is a *particular composition* of the functions. A function has the format as follows:

$$@function\_name[argument_1, argument_2, \ldots, argument_n]; \quad \text{here } n > 0.$$

Note that the symbol @means the beginning of the function and the square brackets "[]"enclose the argument list of the function. An argument in the list may be either a constant, variable (parameter), expression or function. A constant may be either a numeric literal of the language or string literal written in a target language (for example, in VHDL). The constant length in this case may vary from one symbol to the largest fragment. The latter can be a syntactically complete or incomplete. Furthermore, a function is allowed to be incorporated inside of the string literal.

The language has only two types: *string* and *enumerated*. Because they can be recognized from the context, there are no attributes for the explicit declaration of the type. So, the language may be regarded as a *typeless language* (Ousterhout, 1998), this is a feature of the ScL. A value of the string type may be interpreted as a number or text; the decimal and binary numbers can be recognized from the context, too.

We analyse the capabilities of external functions to perform modifications below. These capabilities are: generative, parameterization, internal gluing, computation and control. In Section 5, we will present more advanced capabilities, such as the external gluing.

3.1. *Generative Capabilities*

The language contains a specific function, the *gen* (*generate*) function, that allows to generate "look-alike" strings in a TL. The formal definition of the function is:

$$\begin{aligned}
&@gen\left[< arg1 >, \{< arg2 >\}, \{< arg3 >\}, < arg4 >\right]\\
&< arg1 > ::= < decimal_constant > \mid < variable > \mid < expression >\\
&< arg2 > ::= < string_of_tl > \mid < promol_function > \mid\\
&\qquad\qquad < arg2 >< string_of_tl > \mid < arg2 >< promol_function >\\
&< arg3 > ::= < string_of_tl > \mid < promol_function > \mid\\
&\qquad\qquad < arg3 >< string_of_tl > \mid < arg3 >< promol_function >\\
&< arg4 > ::= < decimal_constant > \mid < variable > \mid < expression >
\end{aligned}$$

The arguments have the following meaning:

- the first argument specifies the number of substrings to be generated;
- the second argument is a "separator" for separating the adjacent substrings;
- the third argument can be considered as an "initial symbol";
- the fourth argument is an "initial extension" for the "initial symbol".

Let us consider the following strings that might be fragments of a particular TL:

$$\mathrm{X1},\ \mathrm{X2},\ \mathrm{X3},\ \mathrm{X4}, \tag{1}$$

$$\mathrm{Y}(1) + \mathrm{Y}(2) + \mathrm{Y}(3), \tag{2}$$

$$\mathrm{A10\ AND\ A11\ AND\ A12\ AND\ A13\ AND\ A14}. \tag{3}$$

These strings can be produced by the adequate descriptions in Open PROMOL as follows:

$$\begin{aligned}
&@\mathrm{gen}\,[4, \{,\}, \{\mathrm{X}\}, 1]\,,\\
&@\mathrm{gen}[3, \{)+\}, \{\mathrm{Y}(\}, 1]),\\
&@\mathrm{gen}\,[5, \{\mathrm{AND}\}, \{\mathrm{A}\}, 10]\,.
\end{aligned}$$

From the first glance there are no advantages in writing these strings in a specific way. The power of that function, however, can be understood if we take into account the following. To produce any string of the type (1)–(3), we need the only *gen* function. For this purpose, we should substitute each *argument-constant* with the *argument-variable* in the *gen* function and use the *sub* function, as illustrated below:

$$@\mathrm{gen}\,[m, \{@\mathrm{sub}[n]\}\,, \{@\mathrm{sub}[p]\}\,, r]\,. \tag{4}$$

The parameter values should be selected as indicated: for m in $\{3, 4, 5\}$; for n in $\{\mathrm{AND},)+, \mathrm{OR}, ,\}$; for p in $\{\mathrm{X}, \mathrm{A}, \mathrm{Y}(, \mathrm{ZZ}\}$; and for r in $\{0..10\}$. As the string (2) has a little difference from the rest, we should write (note that $k = 0$ or 1; see example below):

$$@\mathrm{gen}[m, \{@\mathrm{sub}[n]\}, \{@\mathrm{sub}[p]\}, r]@\mathrm{if}[k = 1, \{)\}].$$

The given fragment of the ScL can be called a *string generator*. Such a generator produces strings of similar structure. The only action should be done before its use: a parameterization in a right way. That means that the syntax of the string can be modified by an external agent (human or program) through parameterization. Each function combined or glued together with the *gen* function can contribute significantly to the generative capabilities of the *gen* function.

3.2. *Parameterization Capabilities*

A function has a list of arguments. An argument may be either the ScL parameter or the TL parameter. The pair of braces "{ }" allows to distinct those parameters types: they always enclose a TL parameter. The ScL parameter may be given as *constant, variable, expression or even as a function of the ScL.* Note that from the standpoint of the ScL concept any TL parameter can't be changed directly. However, a function can bring a particular value of the TL as illustrated by (4). A parameter must have a value to be assigned. For achieving this, we need to add *interface* to the previous example as follows:

```
$
" Enter the length of the string " {3, 4, 5, 6, 7, 8} m := 5;
" Enter the 'separator symbol' for the string" { AND,)+, OR,,} n := AND;
" Enter the 'initial symbol' for the string" {X, A, Y(, ZZ} p := A;
" Enter initial value for the initial symbol extension" {0..10} r := 10;
" Do you need the symbol ')' at the end of the string? (yes=1, no=0)" {0, 1} k := 0;
$

  @gen[m, {@sub[n]}, {@sub[p]}, r]@if[k = 1, {)}]
```

Again, a parameterization in this example seems to be too complicated. Note that this complexity can be hidden and, in some cases, eliminated using such a technique as: default values, automatic parameterization from the context, using the same parameters for different subcomponents, etc.

3.3. *Gluing Capabilities*

A gluing scenario for composition may be very rich because a function allows nesting, i.e., other functions can be inserted as its argument(s) (Fig. 1). Note that any function returns a value that is usually a *modified fragment* of the TL.

This presents a view of the *internal gluing*. As no restrictions exist neither on the length of the target program nor on the number, type and nesting depth of the functions, this can lead to a very sophisticated description presented in two languages as a monolithic structure. It is not an easy task to read and understand such a structure. The latter disadvantage can be eliminated with *external gluing* that uses the external decomposition (the basic principle of structural programming).

The gluing capabilities combined together with the generating capabilities of the *gen* and other functions result in the modification power of the language. We will discuss below other capabilities to pre-program *modifications*.

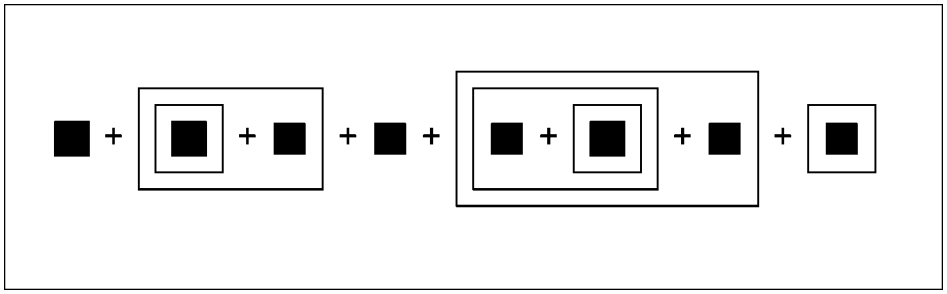

Fig. 1. An abstract representation for gluing: + means gluing, dark rectangle – a piece of target program, a white rectangle – a function of the scripting language.

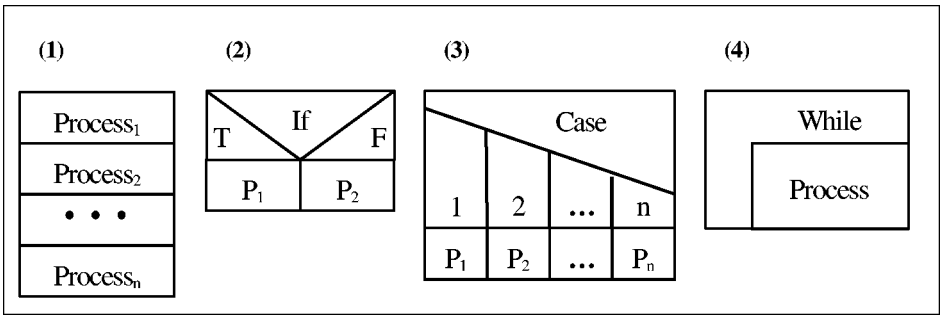

Fig. 2a. Basic structures of structural programming.

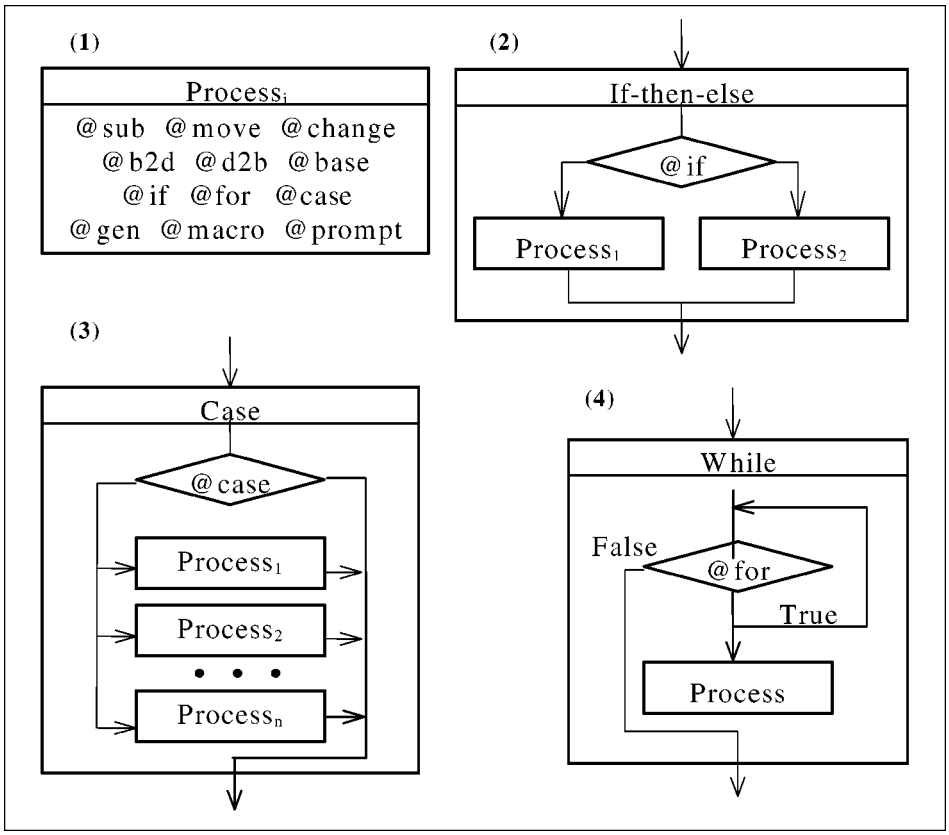

Fig. 2b. Analogy of PROMOL functions and structures of structural programming.

3.4. *Computation and Control Capabilities*

We use functions to develop the target program specifications in the structured programming style. This means that we allow for each function having an argument <*string_of_tl*> an insertion of a function inside that argument. The *nested functions* significantly extend the computation power and allow to modify the fragments of TL flexibly. In Fig. 2a and 2b we illustrate our functions by analogy with the structural programming structures. A computation power depends mainly upon the capabilities of the expressions used in the argument list and the nesting depth of the functions.

We use only one arithmetic type, *integer type*, to describe the parameters and allow the following operations with that type: $+$, $-$, $*$, $/$, $\%$ (remainder), and $\hat{}$ (power). Another type we use is the *string type*. Only the assignment operation is allowed for that type (function *move*). As both types can be easily distinguished from the context, we do not use any attributes to denote those types.

4. The Open PROMOL Processor

The processor is a tool (actually a translator) that supports the use of the scripting language. It has been created using the reuse principles: 1) the processor's main module was produced by **Lex & Yacc** (Manson *et al.*,1990); 2) its output was incorporated into the system as a "component-from-the-shelf" (see Fig. 3).

It should be stated that about 50% of the total amount of the processor's code has been created automatically (using "black-box" principle). The rest 50% part has been created

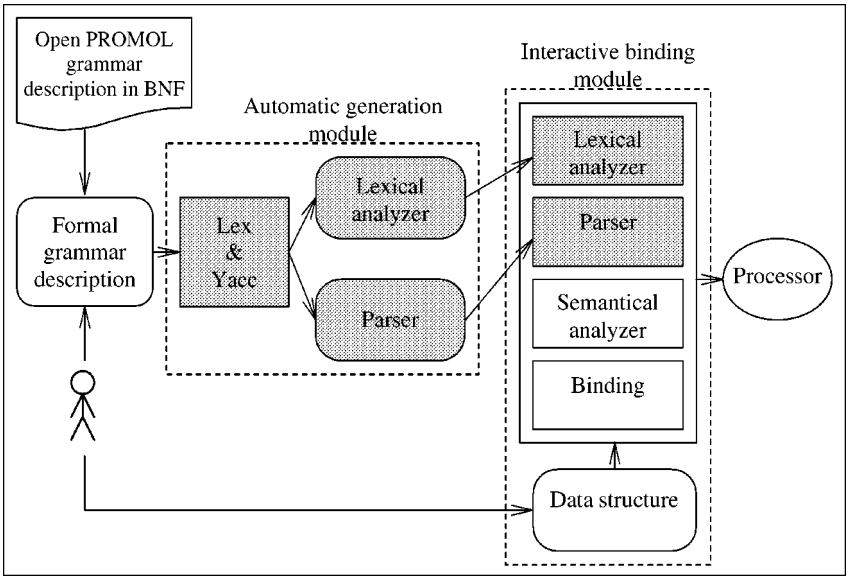

Fig. 3. The processor's development process.

interactively. However, those parts are not equivalent with respect to their complexity. The first part is much more complicated (for the given list of functions (see above, Section 3). The current version contains 358 *grammar rules*, 572 *Finite Machine* S*tates*, 1076 *transitions*, 1827 *token actions*). Furthermore, it is much more reliable than the previous processor due to the high quality of **Lex & Yacc**.

5. An Extended Example: the Internal and External Gluing Capabilities

To illustrate the "gluing" capabilities of the PROMOL functions for modifying a target program, we present two examples. Firstly, we explain the internal gluing. Let be given the VHDL component (instance) that describes the two-input *and-gate* functionality (Fig. 4).

```
ENTITY  GATE  IS
   PORT ( X1, X2 : IN  BIT;
               Y : OUT BIT );
END  GATE;

ARCHITECTURE  BEHAVE_GATE  OF  GATE  IS
BEGIN
   Y <= X1 AND X2;
END  BEHAVE_GATE;
```

Fig. 4. Initial instance in VHDL to be modified.

We need to modify the initial VHDL description according to the prescribed user's requirements that follow below in the gradually increasing order of their complexity.

1. A user needs to have a component with any number of inputs varying from 2 to 8.
2. Additionally to the requirement 1, any functionality must be specified from the list {AND, OR, XOR, NOR, NAND, XNOR}.
3. Additionally to the requirement 2, an extension to the name 'X' must vary from 0 to 10.
4. Additionally to the requirement 3, a user needs to have some flexibility in expressing the input-output delay in the given model. It should be described either explicitly with the generic constant from the list {1, 2, 3, 4, 5 }, or implicitly without that constant (delta delay).

To receive the specification in PROMOL, a designer should perform the following actions:

A. To *map* the informal requirements into the abstract interface specification (see Štuikys e*t al*., 1998) written with the prescribed rules.

B. To *specify* each needed modification with the appropriate Open PROMOL function.

C. To *insert* (glue) the introduced functions into the given VHDL model in a "right" manner.

```
$
"Enter the number of inputs :" {2,3,4,5,6,7,8} num:=2;
$
ENTITY GATE IS
   PORT ( @gen[num,{ , },{X},1] : IN BIT;
          Y : OUT BIT );
END GATE;

ARCHITECTURE BEHAVE_GATE OF GATE IS
BEGIN
  Y <= @gen[num, { AND },{X},1];
END BEHAVE_GATE;
```

Fig. 5. A specification for implementing the requirement 1.

```
$
"Enter a number of inputs :" {2,3,4,5,6,7,8} num:=2;
"Enter a function's name :" {AND,OR,XOR,NOR,NAND,XNOR} f:=OR;
"Enter the initial value :" {0,1,2,3,4,5,6,7,8,9,10} init:=10;
"Do you want to use a generic delay (0-no, 1-yes):" {0,1} use:=1;
use=1 "Enter a delay constant in ns. : " {1,2,3,4,5} delay:=1;
$
ENTITY GATE_@sub[f] IS
  @if[use,{GENERIC ( T : TIME := @sub[delay] NS);}]
   PORT ( @gen[num,{ , },{X},init] : IN BIT;
          Y : OUT BIT );
END GATE_@sub[f];

ARCHITECTURE BEHAVE_GATE_@sub[f] OF GATE_@sub[f] IS
BEGIN
  Y <= @gen[num, { @sub[f] },{X},init] @case[use+1,{ },{AFTER T}];
END BEHAVE_GATE_@sub[f];
```

Fig. 6. A specification for implementing the requirements 1–4.

We illustrate these actions in Fig. 5 and 6, respectively.

The latter example illustrates the use of the so-called "conditional interface", too. To describe the need of generic delay, the parameter *use* has been introduced. This parameter is a part of the following condition in the next question of the interface (see interface in Fig. 6). This question will be given only if the condition (*use* =1) is true. Another important fact evident from this example is the number of modified instances (2,772) that can be generated from the specification. The processor that produces those instances can be regarded as the VHDL "look-alike" model generator. It allows to do more with less.

With the next example we wish to illustrate the external gluing capabilities. Let us consider the two-stage-gate system described in VHDL (see Fig. 7). The requirements for the modification specification are as follows:

1. We need to have a number of gates at the first stage selected from the list {2, 4, 8, 16, 24};
2. A component at this stage must have the same function from {AND, OR, XOR, NOR, NAND, XNOR};
3. A component at the second stage may have any functionality from the above mentioned list.

In Fig. 8 we deliver the solution in PROMOL received in the following sequence.

```
ENTITY  GATE  IS
  PORT (DATA, CNTR : IN BIT_VECTOR (1 TO 2);
        OUTP : OUT BIT);
END GATE;

ARCHITECTURE  EXAMPLE  OF  GATE IS

SIGNAL S : BIT_VECTOR (1 TO 2);

COMPONENT  GATE_NAND_2
  PORT (X1, X2: IN BIT;  Y : OUT BIT);
END  COMPONENT;

BEGIN

  L1 : GATE_NAND_2 PORT MAP (DATA(1),CNTR(1),S(1));
  L2 : GATE_NAND_2 PORT MAP (DATA(2),CNTR(2),S(2));
  L3 : GATE_NAND_2 PORT MAP (S(1),S(2)),OUTP);

END EXAMPLE;
```

Fig. 7. The initial instance of two-stage-nand-gate in VHDL.

a)

```
$                                       @- component : 'this is a comment'
"Enter number of inputs :" {2,4,8,16,24} in:=4;
"Enter the first function :" {AND,OR,XOR,NAND,NOR,XNOR} f:=NAND;
$
COMPONENT  GATE_@sub[f]_@sub[in]
  PORT (@gen[in,{,},{X},1] : IN  BIT;  Y : OUT  BIT);
END  COMPONENT;
```

b)

```
$
@include[component]                     @- to read external component
$
"Enter number of gates at the first stage :" {2,4,8,16,24} in:=4;
"Enter the first stage function:" {AND,OR,XOR,NAND,NOR,XNOR} f1:=NAND;
"Enter the second stage function:" {AND,OR,XOR,NAND,NOR,XNOR} f2:=NAND;
$
ENTITY  SYSTEM  IS
  PORT (DATA, CNTR: IN BIT_VECTOR(1 TO @sub[in]);
        OUTP : OUT BIT);
END  SYSTEM;

ARCHITECTURE  EXAMPLE  OF  SYSTEM  IS
SIGNAL  S : BIT_VECTOR (1 TO @sub[in]);

@macro[component,{2}, {@sub[f1]}]     @- to insert the component
@if[[in=2] and [f1 eq f2],{ },{@macro[component,{@sub[in]},{@sub[f2]}]}]
                                        @- a conditional insertion
  BEGIN
    @for[i,1,in,{
    L@sub[i] : GATE_@sub[f1]_2 PORT MAP (DATA(@sub[i]), CNTR(@sub[i]),
S(@sub[i]));
    }]
    L@sub[in+1] : GATE_@sub[f2]_@sub[in] PORT MAP (@gen[in,{},},{S(},1]),
OUTP);

END  EXAMPLE;
```

Fig. 8. External gluing: a) external component; b) gluing capabilities.

```
ENTITY SYSTEM IS
 PORT (DATA, CNTR : IN BIT_VECTOR (1 TO 4);
     OUTP : OUT BIT);
END SYSTEM;

ARCHITECTURE EXAMPLE OF SYSTEM IS
SIGNAL S : BIT_VECTOR (1 TO 4);

COMPONENT GATE_NAND_2
 PORT (x1, x2 : IN BIT; y : OUT BIT);
END COMPONENT;

COMPONENT GATE_NAND_4
 PORT (x1, x2, x3, x4 : IN BIT; y : OUT BIT);
END COMPONENT;

 BEGIN

  L1 : GATE_NAND_ 2 PORT MAP (DATA(1), CNTR(1), S(1));

  L2 : GATE_NAND_2 PORT MAP (DATA(2), CNTR(2), S(2));

  L3 : GATE_NAND_2 PORT MAP (DATA(3), CNTR(3), S(3));

  L4 : GATE_NAND_2 PORT MAP (DATA(4), CNTR(4), S(4));

  L5 : GATE_NAND_4 PORT MAP (S(1), S(2), S(3), S(4), OUTP);

END EXAMPLE;
```

Fig. 9. An instance in VHDL produced with PROMOL processor from the description given in Fig. 8.

Firstly, we have developed a generalised subcomponent: *component.* Secondly, we have described reading of the *component* with the *include* function. Note that the parameter of the function is the subcomponent name (file name). And finally, we have developed the system for gluing the external subcomponent with *macro* and other functions.

As a result, we can produce 180 different instances from that model using PROMOL processor. We present the generated instance in Fig. 9 (this example corresponds to the description in VHDL of the circuit given in Milne's book, page 37 (Milne, 1994)).

6. Preliminary Experiments with the Open PROMOL Processor

Preliminary experiments with the processor include 23 tests developed for teaching and testing purposes, VHDL packages for multiplexer, generation of the formal description in BNF of Open PROMOL subsets, data types generation for C++, Pascal and Java. With those experiments we receive an approval that our processor is the target language-independent.

7. Evaluation of the Results: Comparison with Other Approaches

We have compared our results with the cases known from the literature (see Table 1). The different authors use slightly different approaches for achieving the same aim: component generalization. What is new in our approach is that our language is built on the external function concept, and we have suggested specific functions. Another distinction is in the component vision. Components, either the *syntactically complete* or *incomplete* structures, are the building "bricks" for composition. The functional capabilities of our approach are similar to those given in the literature. For example, we regenerated the examples described in Bassett's book (Bassett, 1997) with our functions easily.

Table 1

The comparison of generic (reuse) component development technologies

Supporting Tools	Model	Basic complexity (command number)	Implementation characteristics of the component	Language independent (Y) or dependent (N)	Application domain for approval
Preprocessor	External functions	8	Monolithic component	N	VHDL models
Open PROMOL Processor	PROMOL functions	13*	Structural component	Y	VHDL and other TL models
Frame commands processor (Bassett)	Frame commands	>5	Frame composition	Y	Commercial systems
P++ translator (Batory, *et al.*)	GenVoca model (P++)	Unknown	Component's realm	N	Data structures & others
Extended C++ processor (Arney)	TCL commands	About 30	Reuse component	N	Universal (non-specified)

The first finding that we received from the preliminary experiments is the processor's reliability. This has been achieved due to the **Lex & Yacc** module, the "component-from-the-shelf". The processor is a system that generates the "look-alike" program models in a given TL from the specification that describes the needed modifications. The processor allows to do more with less. It seems that it is not an easy task to develop the models (to pre-program modifications) using two languages at the same time. We should have in mind that an instance of a target program is always given. This instance must be syntactically and semantically correct. It is easier to introduce changes (if they are well-specified) than to develop the model from scratch. Of course, some experience of working with the Open PROMOL functions is needed.

The second finding is the system openness. This regards to the language and its processor. As syntax of the language is well-defined and simple, it is an easy task to introduce new functions (for implementing new applications more efficiently) and re-program the processor, because we apply reuse methodology.

The third finding is the language capabilities for modifying and gluing pieces of a target language. Each function returns a value that is usually the modified fragment of a target language. As a scripting program is a composition of the functions, gluing takes place in a natural way automatically. Modifying capabilities depend on a parameterization. As a parameter of a function may be a constant, variable, expression, other function and *even name of a component*, modifications can be performed flexibly.

8. Concluding Remarks

The proposed language is a tool for developing of the generalized components. The language supplies capabilities for composing a target program, too. The external functions of the language allow to perform a composition in a simple and easy way. We are convinced of the independence of the proposed script language from target languages.

The Open PROMOL processor can be considered as a generative tool for generating "look-alike" programs in a target language. It allows to do more with less. The reliability of the processor rely on the use of **Lex & Yacc** as a tool for developing the processor.

This work and recent works of other authors convinced us that generative approach should be based on the relatively small generic library. Its components should be accessed through processor. This work should be regarded as a case study in this direction.

We formulate the problems for further consideration as follows:

1) Applications for the language and its processor;

2) Various generator models based on the use of Open PROMOL and its processor.

Acknowledgements

Authors would like to thank to the anonymous reviewer whose notes served for an improvement of the paper.

References

Arney, J. S. (1998). C Preprocessing with TCL. *Dr. Dobb's Journal*, August, 46–49.

Bassett, P. G. (1997). *Framing Software Reuse: Lessons from the Real World*. Prentice Hall, Inc.

Bassett, P. G. (1999). Is Reuse a Transient Issue? *Component Strategies*, January, 64.

Batory, D., V. SinGhal, J. Thomas, S. Dasari, B. Geraci, M. Sirkin (1994). The GenVoca model of software system generators. *IEEE Software*, September, 89–94.

Batory, D., V. SinGhal, J. Thomas, S. Dasari, B. Geraci, M. Sirkin (1995). Achieving reuse with software system generators. *IEEE Software*, September, 89–94.

Batory, D., B. J. Geraci (1997). Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering*, **2**(23), 67–82.

Batory, D., B. Lofaso, Y. Smaragdakis (1998). JTS: Tools for implementing domain-specific languages. *Proceedings of 5th International Conference on Software Reuse*, IEEE Computer Society, 143–153.

Brown, A. W., K. C. Wallnau (1998). The current state of CBSE. *IEEE Software*, September/October, 37–46.

Broy, M., A. Deimel *et al.* (1998). What characterizes a (software) component? *Software – Concepts & Tools*, **19**(1), 49–56.

Czarnecki, K., U. Eisenecker, R.Gluck, D. Vandervoorde, T. Veldhuizen (1999). Generative Programming and Active Libraries (Extended Abstract), `http://extreme.indiana.edu/tveldhui/papers/dagstuhl1998/dagstuhl.html`.

Chang, K.C. (1997). *Digital Design and Modeling with VHDL and Synthesis*. IEEE Computer Society Press.

Hudak, P. (1998). Modular domain specific languages and tools. *Proceedings of 5th International Conference on Software Reuse*, IEEE Computer Society, 134–142.

IEEE Std. 1076 (1996). *VHDL Interactive Tutorial: A Learning Tool for IEEE Std. 1076 VHDL.* IEEE, CD-ROM.

Jacobson, I., M. Griss, P. Jonsson. (1997). *Software Reuse (Architecture, Process and Organization for Business Success)*. Addison-Wesley.

Kozaczynski, W., G. Booch. (1998). Component-based software engineering. *IEEE Software*, September/October, 34–36.

Manson, T., D. Brown (1990). *Unix Programming Tools. Lex & Yacc.* O'Reilly & Associates, Inc.

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

Milne, G. (1994). *Formal Specification and Verification of Digital Systems*. McGRAW HILL Book Company.

Ousterhout, J. K. (1993). *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company.

Ousterhout, J. K. (1998). Scripting: higher level programming for the 21^{st} century. *IEEE Computer*, **31**(3), 23–30.

Pfleeger, S. L. (1998). The nature of system change. *IEEE Software*, May/June, 87–90.

Rembold, U., R. Dillman. (1986). *Computer-aided Design and Manufacturing (Method and Tools)*. Springer-Verlag.

Sametinger, J. (1997). *Software Engineering with Reusable Components*. Springer.

Schneider, J.G., O. Nierstrasz (1999). Components, scripts and glue. In Barroca L., J. Hall and P. Hall (Eds.), *Software Architectures – Advances and Applications*, Springer, pp. 13–25.

Štuikys, V. (1998). Design of reusable VHDL component using external functions. *Informatica*, **4**(9), 491–506.

Štuikys, V., O. Olsen, G. Ziberkas. (1998). Building of VHDL reusable components for DSP-oriented architectures. *Information Technology & Control*, **3**(9), 43–53.

Terry, P. D. (1997). *Compilers and Compiler Generators: an Introduction with C++*. International Thomson Computer Press.

V. Štuikys received Ph.D. degree from Kaunas Politechnic Institute in 1970. He is currently Associate Professor at Computer Department, Kaunas University of Technology, Lithuania. His research interests include domain-specific reuse, high level domain-specific languages, expert systems, digital signal processing and CAD systems.

R. Damaševičius received a bachelor degree in informatics from Kaunas University of Technology in 1999. He is currently MSc student at Informatics faculty, Kaunas University of Technology. His research interests include software reuse, scripting and programming languages, development of kits for supporting reuse.

Scenarijų kalba Open PROMOL ir jos procesorius

Vytautas ŠTUIKYS, Robertas DAMAŠEVIČIUS

Straipsnyje aprašomos scenarijų kalbos Open PROMOL ir jos procesoriaus galimybės. Ši kalba yra skirta specifikacijoms (scenarijams) kurti, kai norima modifikuoti kita išorine kalba užrašytus programų tekstus. Mes vartojame šitos kalbos procesorių kaip įrankį kurti savarankiškus atsikartojančius komponentus arba kaip programų generatoriaus modulį ("komponentą iš lentynos"). Pačiame procesoriuje yra įterptas Lex & Yacc sugeneruotas modulis, kitas žemesnio lygmens "komponentas iš lentynos". Mes pateikiame šitokias siūlomos kalbos ir jos procesoriaus galimybes: generavimo, skaičiavimo, valdymo, parametrizavimo ir "klijavimo" (komponavimo). Mes lyginame siūlomą metodą su panašiais literatūroje aprašytais metodais.