INFORMATICA, 2000, Vol. 11, No. 3, 325–348
© 2000 *Institute of Mathematics and Informatics, Vilnius*

The Language-Centric Program Generator Models: 3L Paradigm

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

Abstract. In this paper we suggest a three-language (3L) paradigm for building the program generator models. The basis of the paradigm is a *relationship model* of the *specification*, *scripting* and *target languages*. It is not necessary that all three languages would be the separate ones. We consider some *internal relationship (roles)* between the capabilities of a given language for specifying, scripting (gluing) and describing the domain functionality. We also assume that a target language is basic. We introduce domain architecture (functionality) with the *generic components* usually composed using the scripting and target languages. The specification language is for describing user's needs for the domain functionality to be extracted from the system. We present the framework for implementing the 3L paradigm and some results from the experimental systems developed for a validation of the approach.

Key words: specification language, scripting language, target language, generic component, program generator, generative reuse, application domain, domain-specific language, VHDL.

1. Introduction

Program generation is the process of producing code automatically by a *program generator*. In contrast to a compiler, a generator *usually* produces code in a *high-level target language*. The target language may be either the *conventional* or *domain-specific language*. In recent years the interest grows in the program generators increasingly. The reason is that future of software engineering lies in the automated development of well-understood software in order to keep pace with the ever-increasing productivity in the development and production of computer hardware.

It is a non-trivial task to build a generator or generating system for several reasons. First, a generator is usually applied in a domain-specific area. A generator can't be build without understanding of its domain. It is not sufficient to propose a domain model. The model must be validated through extensive experiments and prototyping. Next, a generator can't be considered as a stand-alone system only, but as a subsystem of the integrated system, too. Finally, for the generalisation purposes and accessibility of the results for a wider community, the tool design principles must be identified and represented in a domain-independent way. The generalisation can be achieved by introducing some formalism and abstraction. This requires more deep domain knowledge and efforts from the designer than building a system from scratch.

Program generators were already mentioned twenty-five years ago (or so) in the literature. From the perspective, now we can observe an evident shift: 1) from the *template-centric* to *component-based* models; 2) from the *domain-dependent* to *domain-independent* models; 3) from the *fine-grained* to *course-grained components;* 4) from the *stand-alone implementations* to *implementations linking domain analysis and architectures* with software generators.

In this paper we suggest a *three-language (3L) paradigm* for building the program generator models. The paradigm basis is a *relationship model* of the *specification*, *scripting* and *target languages*. However, it is not necessary that all three languages would be the separate ones. We describe boundaries, roles and relationship of these languages from the viewpoint of building a generating system for a given domain. With respect to the control and transformation capabilities, our models are *abstract machines* that can be interpreted as *generalised translators for higher-order programs*. We present some results of experiments gained from the experimental generating systems developed for a validation of the approach.

The structure of the paper is as follows. We present the review of the related works in Section 2. We describe our approach in general in Section 3. We present the details of our 3L paradigm in Section 4. Some implementation results are given in Section 5. We describe the experimental results in Section 6. Finally, we present summary, evaluation of the results and concluding remarks in Sections 7 and 8, respectively.

2. Review of the Related Works

Program generators are the invention of about twenty-five years old. According to our knowledge, perhaps, the first referencing to the problem has been made by several authors in the area of compiler generators (Johnson, 1975; Lesk, 1975; *see also* Terry, 1997). The domain-specific generators have been in use in some specialised areas, such as discrete event simulation (Luker *et al.*, 1979), hardware testing (Tinaztepe *et al.*, 1979; Oswald *et al.*, 1983), teaching of programming (Teitelbaum *et al.*, 1981). The main intention of their usage was to increase the programming productivity for inexperienced users. Several years later, CASE (Computer-Aided Software Engineering) tools (Terry, 1987) and application generators in data processing area (Phelps, 1987) became available as a market product for workstations and mainframes. Luker and Burns summarise the achievements of the first decade in (Luker *et al.*, 1986). Their work focuses on the generalised structure (generator's model) and user's dialogue (the specification problem).

Program generators have received an increased attention at the end of 90's with the development of research in reuse. **Lex & Yacc**, perhaps, is the best-known example of a program generator (Mason *et al.*, 1990; Levine *et al.*, 1992). The initial understanding of the up-to-date generator models can be gained through analysis of the tool definitions. "*A generator creates derived components and various relationships from languages and templates. The generator is often a conversational or language-driven tool*" (Jacobson *et al.*, 1997). Lim (1998) presents other view: a generator is "*a higher-level automatic* *builder that hides the manual interconnection of components using a problem-oriented language, template or option filter, or a visual environment*". Furthermore, the author adds that "*the generator enables concise specification of the desired (piece of) the application, and then generates code and/or procedure calls in some other language*".

At the abstract level, many authors conceive a generator model as a *process* that uses (transforms) two structures (parts): the *invariant* and *variant ones*. The invariant part *is reused* without changes. The variant part serves for adding a functionality to *customise* the invariant part. The central problem is how the invariant part should be represented. A formalism or abstraction used to represent the invariant part varies in a wide range: from the simplest templates, such as language structures with "holes", i.e., omitted non-terminals (Teitelbaum *et al.*, 1981) to more sophisticated templates, such as used in skeleton or kitchen sink approaches (Sametinger, 1997); from the *generics and packages* in domain-specific languages, such as VHDL (Ashenden, 1996; VHDL Tutorial, 1996) to *domain-specific language* itself; from the language patterns, such as assembly language to object-oriented abstraction in such language as C++ (Sametinger, 1997); from the programming *language extensions* (e.g., P++ is the C++ extension in GenVoca model (Batory *et al.*, 1995)) to *scripting languages,* such as TCL (Ousterhout, 1993).

Perhaps, there is no universal abstraction that suits best for most practical purposes, i.e., for building reusable components and generators. However, the known abstractions might be categorized into two large categories with respect to the target language as follows: the *internal* and *external* abstractions. By the internal abstractions we mean those that exist in a given language we use to describe the component (generics, packages, abstract data types, mechanisms that support object-orientation and other kind of generalisation, etc.). By the external abstractions we mean those that are or might be introduced as an extension of a given language (scripting languages, the above-mentioned Batory's P++, etc.). The external abstraction is called a *meta-program,* too (Batory, 1998a). The template abstraction (aka skeleton, frame; for discrepancies in terminology see in (Sametinger, 1997; Bassett, 1997)) might be interpreted either as an internal or external abstraction. That depends upon what is the nature of the template: if it is a part of the language itself, for example, *PASCAL(PL/CS)*_structures with "holes" (Teitelbaum *et al.*, 1981), *ATLAS*_frames (Oswald *et al.*, 1983; Štuikys *et al.*, 1993) may be considered as the internal abstractions. However, FSM[1] templates in the form of state tables are introduced into the generating system from outside (Štuikys *et al.*, 1997) and should be treated as an external abstraction.

Furthermore, some formalism such as HOL (Higher-Order Logic) is powerful enough and can be used as a tool for implementing the abstraction independently (Milne, 1994). Other abstractions such as scripting languages are supplementary for target languages. The role of scripting (aka gluing, composition) languages for reuse was pointed out by several authors (Griss, 1993; Ousterhout, 1998). The scripting language may be interpreted as a higher-order language aiming to glue subcomponents developed in other language into a system. The latter may be either a domain-specific or conventional language.

[1]Finite State Machine

The capabilities of scripting languages vary from simple to the sophisticated ones such as TCL (Ousterhout, 1998; Schneider *et al.,* 1999).

At the lower level of a generating model analysis, one should conceive that the invariant part is *more related with a generic library,* and the variant part *is more related with a specification module* of a *generator*. Actually the invariant part must be treated as a set of *generic components,* the items of a generic library. The *library component-centric generation models* as well as *tools or frameworks* for implementing generators itself might be recognized as perspective directions in generative reuse (Thibault *et al.,* 1997; Batory *et al.,* 1998; Biggerstaff, 1998; Czarnecki *et al.,* 1999). To build a generic component or generator, first we need to extract *commonalties*, *differences* and *similarities* from a domain. The activity that serves for achieving this aim is called domain analysis (DA). The primary goal of DA is to build a domain model. The most general domain model is a *domain language* (Prieto-Diaz, 1990; Iscoe *et al*., 1991). Many different approaches exist how to perform DA (Batory, 1998b; Neighbors, 1998). The best known example is FODA, the Feature-Oriented DA method (Sametinger, 1997). A domain can be treated as well-understood one, if a maturity level of domain knowledge is high. That level can be measured, for example, by the number of standards, their status and evolution, the number of systems and their maturity, the quality and population of products, the efforts and support from industry and governments, the number of subdomains, the number of publications existing in (sub)domains, etc. Among others, the applications related to hardware design with the standard high-level domain language VHDL (VHDL Interactive Tutorial, 1996) must be mentioned.

For well-understood domains, such as mentioned above, a domain language already exists. According to Pietro-Diaz (Prieto-Diaz, 1990) this is the "*reuse of analysis of information*" and it is the "*most powerful sort of reuse*". The standard high-level language yields a very broad knowledge for understanding of the domain. As a result, the domain can be subdivided into subdomains and applications. For example, the standard VHDL can be split into subsets such as VITAL (Chang, 1998) for more narrow applications. The domain content is very important for successful reuse. Biggerstaff (Biggerstaff, 1998) has reported the argument that "*the first order term in the success equation of reuse is the amount of domain-specific content and the second order term is the specific technology chosen in which to express that content*". A *domain-specific language* (DSL) allows to express better the domain content and implement reuse rather than conventional programming language (Thibault *et al,* 1999; Hudak, 1998). This is the first reason why DSLs have attracted the increased attention in recent years. The second reason is the less cost at the latest life cycle phase of a software system implemented with DSL (Hudak, 1998). Both languages (the conventional and DSL), however, may reside in the same integrated system as follows from the next analysis.

Usually a representation of the target system (application) in a DSL is not a self-aimed product. For domain-specific applications, such as real time systems (ATLAS-related applications (IEEE standard ATLAS Test Language, 1981), partially VHDL-related applications), the target product is the lower-level control data (program) to control the production line process (instruments, equipment). In this case a DSL serves for interfacing between the higher-level and lower-level specifications. However, the latter is implemented using the conventional language (e.g., C or C++) and its environment (ARINC Specification 608–1, 1989; Štuikys *et al.*, 1993).

Our research in automatic program generation is related with some domains, including VHDL domain. We analyse how the language supports the following capabilities in our generator models:

a) the representation of the domain object functionality in some generalised manner (generic component design);

b) the scripting of the related functionality in order to express the higher-order functionality (variations);

c) the specifying of a domain problem when generating its abstract description in the target language;

d) the composition of the language abstractions with abstractions of the external language Open PROMOL (Štuikys *et al.*, 2000) for the purposes of generalisation.

The context in which the different subsets of the same language (VHDL in our case) can play a role of scripting, specification and describing the given functionality at the same time is a very important feature of the approach proposed. Its details are as follows.

3. A General Description of the Approach

We introduce a paradigm for building the program generator models. At the abstract level, we describe our approach using the following concepts: the *target language (TL), specification language (SpL), scripting language (ScL), generic component (GC), generic component library (GCL)* and *abstract machine* for *transformation and control* (see Fig.1). Each concept has a particular role and relates to the abstract "building blocks" that may be evolved for representing the models at the lower level. As the term "language" plays the essential role in this case, we call our model the *language-centric* model. Our approach is based on two assumptions.

The *first* assumption is that a generating system should perform at least three basic functions:

a) to allow specifying a domain problem;

b) to allow representing domain functionality in some generalised manner;

c) to produce a target system with respect to a given specification.

The *second* assumption is that three languages may describe these functions: the *target*, *scripting*, and *specification*. 3L paradigm doesn't mean that these languages are necessarily the separate and independent ones. Either the different subsets of a given language can play a role of a target, scripting and specification language, or two separate languages combined into a system in a appropriate way can enhance these capabilities.

However, the TL is always assumed as basic. The rest two languages are complementary to the basic one. In other words, our paradigm is based on a particular relationship between these three languages with a particular role of the TL. This relationship exists because there is no "pure" TL. Each language that we use to express domain functionality also contains some capabilities for gluing the different functionality and specifying a

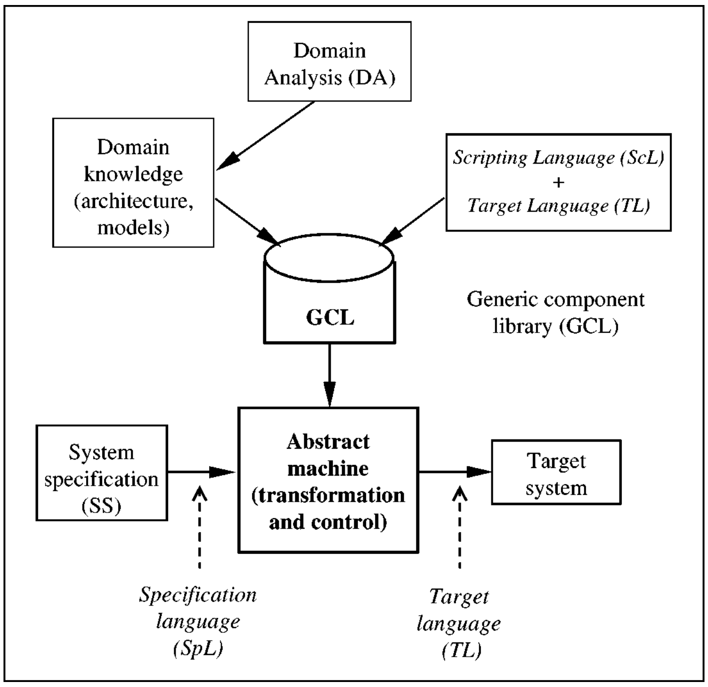

Domain Analysis (DA)
Domain knowledge (architecture, models)
Scripting Language (ScL) + Target Language (TL)
GCL
Generic component library (GCL)
System specification (SS)
Abstract machine (transformation and control)
Target system
Specification language (SpL)
Target language (TL)

Fig. 1. The paradigm of a generating system.

domain problem. Composing in an appropriate way a system, say, of two different languages (one of them is a TL), we can enhance the specification and scripting capabilities of a TL in a desirable manner for generating purposes. The problem we deal with is formulated as follows:

1) how to capture a relationship between capabilities of a TL to express a domain functionality;
2) how to describe scripting of the different functionality for generalisation purposes and specify a domain problem (how to built the relationship model);
3) how to enhance this model introducing extra capabilities with an external language;
4) how to explore and use this model for building generating systems.

It is a conventional way to specify a domain problem (the input to the system) by writing a specification. We assume that a SpL serves for this purpose. We use this term here in a narrow sense having in mind the abstract but not formal specifications (such as HOL, Z or VDM). Furthermore, we assume also that SpL may be a subset of a given target or scripting language. The yield of the system is a target program. A TL is for coding a target program that describes the domain functionality. Later, at the lower level of the model description, we assign a particular meaning to both the TL and SpL.

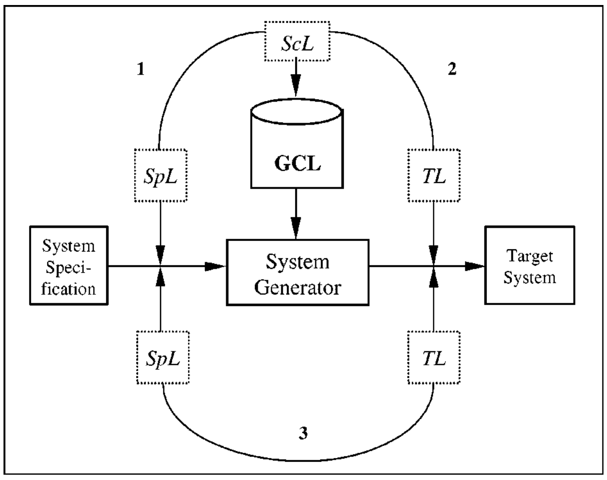
A component (an instance) is a black-box entity that encapsulates a service or a prescribed functionality behind the well-established interface. The GC (aka reusable component) encapsulates a set of the component instances. A concrete instance can be derived in a particular way from that GC. Since a GC is some generalisation, the number of GC's is much less than the total number of instances. No matter how large or small is a collection of GC's used in the generative system, we need to have a library, called the GCL, for keeping them.

A ScL is a tool for developing and representing either GC's or a system composed from GC's (in other words, either the *fine-grained* or *coarse-grained* GC's). Input data for developing a GC is a domain knowledge gained as a result of domain analysis (DA). A structure of a GC may be conceived of as a *concise* collection of the component instances. An instance of a component represents an algorithm or functionality of the target domain. We assume that a TL is for describing this functionality. The aim of GC's is to introduce the target domain (application) algorithms in a generalised way for the generating system. In other words, a GC is a concise description of the different functionality delivered as a logically complete structure. A GC may be either the *generalised instance* or *a composition* of instances. Before the development of GC's, the application algorithms should be disclosed and understood. This is a matter of DA. We assume that algorithms can be described in a domain-specific language (DSL), as well as in a conventional programming language (the TL in this context). In our model a role of a ScL is either to *generalise or compose* the component instances for developing GC's. In general, this means that a ScL is not the self-contained. It should be used together with a TL. A TL serves for describing target algorithms while a ScL serves for composition and generalisation. However, we assume that some "overlapping" of the roles of those languages may exist, as it will be shown in the next Section.

4. The Relationship Model between Languages in 3L Paradigm

In the previous Section we have pointed out a relationship between the TL and ScL. Below we present a more accurate model. This relationship model comprises all three languages and their features. Note that we analyse this model in the narrow sense here, i.e., only from the viewpoint of building of the generating systems. One important aspect of the model (see Fig. 2, *a*) is that there may be either a specific or advanced language(s) which (whose) different subsets (or separate constructs) can play a role of the TL, ScL and SpL at the same time (see in Fig. 2, *a* the relationship chains 1, 2 and 3). We detail this model feature in Fig. 2, *b*. We present the roles of three languages in the right-hand side of the table. The left-hand side of the table illustrates concrete instances (names) of the languages that we analyse in the role of the TL, ScL and SpL while implementing a generating system.

One can assume (see Fig. 2, *b*) that **Open PROMOL** (this is a full title of the language) and its processor play the central role in our approach. **Open PROMOL** (**PRO**gram **MO**dification **L**anguage) is the scripting language aiming to specify modifications of a program written in a TL. The language has been developed as an experimental language to support the generic (reusable) components and generating systems design. The main concept of the language is a *set* of *external functions,* initially described



a)

Variants (cases) for implementing (testing) the conceptual model			Roles of languages		
			SpL	**ScL**	**TL**
1) TL≡PROMOL;	**ScL≡PROMOL;**	**SpL≡PROMOL;**	TL	TL	TL
2) TL≡VHDL;	**ScL≡VHDL;**	**SpL≡VHDL;**	TL	TL	TL
3) TL≡VHDL;	**ScL≡PROMOL;**	**SpL≡PROMOL;**	ScL	ScL	TL
4) TL≡C++;	**ScL≡PROMOL;**	**SpL≡PROMOL;**	ScL	ScL	TL
5) TL≡PASCAL;	**ScL≡PROMOL;**	**SpL≡PROMOL;**	ScL	ScL	TL
6) TL≡BNF;	**ScL≡PROMOL;**	**SpL≡PROMOL;**	ScL	ScL	TL
7) TL≡VHDL;	**ScL≡PROMOL;**	**SpL≡VHDL;**	TL	ScL	TL
8) TL≡JAVA;	**ScL≡PROMOL;**	**SpL≡PROMOL**	ScL	ScL	TL
9) TL≡CLARION;	**ScL≡PROMOL;**	**SpL≡PROMOL;**	ScL	ScL	TL

b)

Fig. 2. Relationship of target, specification and scripting languages (a) and variants of the model (b).

in (Štuikys, 1998). This set is still *open* for extensions that might arise due to the new applications to be introduced. We argue that the component generalisation can be achieved with Open PROMOL through the external pre-programmed modifications of the given target program.

For more than two years the language and its processor is exploited by post-graduate students as an experimental tool. As PROMOL processor has been developed using **Lex** & **Yacc**, we can support a rapid prototyping and carry out continuous improvements and extensions on both platforms, the Win32 (Windows NT32) and UNIX (SunOS). However, the main concept of the language has been preserved during this process because *external functions* are a powerful abstraction allowing to perform a composition for modification and generalization in a simple and natural way. A function always returns a string of the TL that is being composed with its context in some pre-programmed way. The scripting program is a *particular composition* of the functions. A function has the format as follows:

$$@\mathit{function\_name}[\mathit{argument}_1,\ \mathit{argument}_2, \ldots,\ \mathit{argument}_n]; \quad \text{here } n > 0.$$

Some functions such as @***if***, @***case***, @***for*** can be found as control structures (operators) in the conventional programming languages under different or similar names. However, other functions such as ***@gen*** (this function serves for generating the "look-alike" strings in a TL) are specific for Open PROMOL.

The language has actually one type, the *string* type. However, this type can be interpreted in different contexts differently (either as a *text*, *integer*, *real, binary* or *enumerated* type). Because this difference 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*, this is a feature of the ScL (Ousterhout, 1998).

An example that illustrates the PROMOL usage for representing *explicitly* the multiplication of matrixes follows below (see Fig. 3). The PROMOL functions are integrated together with the symbols of a TL as follows: $C[\,] = A[\,] * B[\,]+;$. As a result of interpretation of the program and taking into account that ($r1 = 2;\ c1 = 2;\ r2 = 2;\ c2 = 2$), we will receive:

$$C[1,1] = A[1,1] * B[1,1] + A[1,2] * B[2,1];$$
$$C[1,2] = A[1,1] * B[1,2] + A[1,2] * B[2,2];$$
$$C[2,1] = A[2,1] * B[1,1] + A[2,2] * B[2,1];$$
$$C[2,2] = A[2,1] * B[1,2] + A[2,2] * B[2,2];$$

From the architectural (structural) viewpoint, a PROMOL specification (aka component) can be represented in two different ways. The first representation is a single specification module as given in Fig. 3. The second one is a set of interrelated external modules (see Štuikys, 2000). A single module consists of two parts: the interface program and specification body (implementation).

We distinguish two points to the specification with respect to the development and usage phases: *the designer's viewpoint* and *user's viewpoint*. The specification's designer implements the requirements for modification and generalization using the PROMOL language and its processor. He (she) implements a specification as a generic component in a whole. The implementation covers the statement of requirements, the specification development (analysis of a domain model, mapping of a given set of requirements into set of parameter values, i.e., coding of the interface program, and coding the specification body with external functions), testing and certification. A user usually works with the interface program and produces instances from the specifications developed by a designer. From the user's viewpoint, the interface program is *visible*, and the specification body is *hidden*. This is an ideal case. Of course, a user can modify, if needed, the previously

```
$
      "Specify the number of rows in the first matrix:"          {2..10}   r1:=10;
      "Specify the number of columns in the first matrix:"       {2..10}   c1:=10;
      "Specify the number of rows in the second matrix:"         {2..10}   r2:=10;
      "Specify the number of columns in the second matrix:       {2..10}   c2:=10;
$
@if[c1/ = r2, {
Matrix multiplication is impossible.
},{
@for[i, 1, r1, {
      @for[j, 1, c2, {
C [ @sub[i], @sub[j] ] = @for[k, 1, c1,
{A [ @sub[i], @sub[k]]* B [ @sub[k], @sub[j]]@if[k/ = c1, {+}, {;}] }
                     ] @– the end of the second inner for function
           }] @– the end of the first inner for function
        }] @– the end of the outer for function
    }] @– the end of the if function
```

Fig. 3. PROMOL specification represents explicitly the multiplication of two matrixes in a TL.

developed specification without any restrictions. This view is very important because the *interface program* can be understood as a *problem specification part* for the processor.

In Fig. 4 we deliver a generalized template of an interface program (compare it with the one given in Fig. 3). It describes a domain problem specification. The following is new in this template: a) the **include** statement for declaration of external *promol_*modules which might be used in the specification; b) *conditional* assignment that is executed only if the given condition (represented by a logical expression) has value *true* (or 1) (note that the condition must be expressed by parameters which values have been already specified). By the *specification capabilities* of PROMOL we mean those we use to code *interface programs*.

As has been stated previously, a specification can possess a hierarchical structure consisting of the interrelated external modules. In this case interface program also has the hierarchical structure, i.e. each module has its own interface program. We summarize analysis of PROMOL and its specification as follows.

The modification (generalisation) power and scripting flexibility of the language mainly depends upon:

a) *parameterization and specification capabilities* (the language implements the multi-level parameterization concept with respect to the target program: external file (module) name - *promol_*function itself - argument of the function - a nested function as an argument of the given function (see Fig. 4 and 3));

b) *a computation power* (the language allows the main arithmetic, relational and logical operators, expressions for the above mentioned operators, standard computation functions such as *cos*);

```
$
@include[< module_name_1 >, ... ,< module_name_k >]
$
   "A question_1 related with the parameter_1 of the application:"
   {< list_of_values_of_parameter_1 >} < parameter_1 >:=< value_from_list_1 >;
...                ...                   ...
[<condition>] "A question_i related with the parameter_i of the application:"
   {< list_of_ values_of_parameter_ i >} < parameter_i >:=< value_from_list_i >;
...                ...                   ...
   "A question_n related with the parameter_n of the application:"
   {< list_of_values_of_parameter_n >} < parameter_n >:= < value_from_list_n >;
$
```

Fig. 4. PROMOL interface program template for specifying a domain problem.

c) capabilities to manipulate with a unique data type, the *string* type, and recognise other types (*integer*, *real*, *bit*) from the context;

d) capabilities of nesting (an argument of a particular function may be other PROMOL function; there are no constrains on the nesting deepness if the nesting is allowed for that argument);

e) the capability to compose external PROMOL modules;

f) usage of the special editor **PROMed (PROM**OL **ed**itor**)** that allows syntax highlighting and in this way to overcome partially difficulties caused by the *prettyprinting problem;*

g) the independence of the PROMOL processor from the target language.

As has been stated previously, ScL (PROMOL in our case) is used together with the TL. The standard VHDL (VHDL, Interactive Tutorial , 1996) stands for this purpose in VHDL-related applications. We distinguish the following language capabilities (roles): a) to express domain functionality (i.e., to describe domain algorithms in the conventional way); b) to represent either a system as a set of scripts (subcomponents) or express the domain functionality in a more general way; and c) to specify the target system with the specification capabilities of the language. In other words, we consider the standard as a language consisting of two interrelated (overlapped) *subsets*: the *structural subset* and *functional subset*. At the highest level, for example, we specify the target system with the structural subset.

We can summarise the main result of the previous analysis as follows (see also Fig. 5):

1) In order to develop GC's, we can use the following abstractions: *external* PROMOL functions, the *internal* abstractions of the TL, and the *mixed* abstractions (external functions and internal capabilities of a TL);

2) We can cover a family of instances of domain models only with a few GC's;

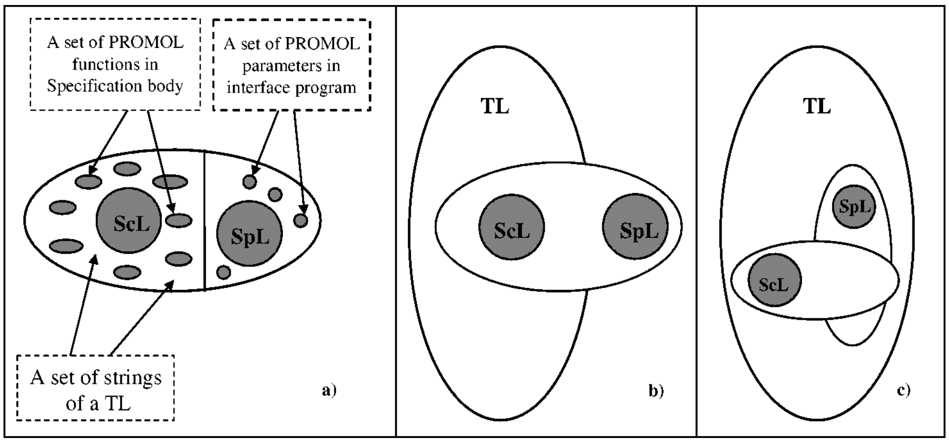

A set of PROMOL functions in Specification body
A set of PROMOL parameters in interface program
ScL
SpL
A set of strings of a TL
a)
TL
ScL
SpL
b)
TL
SpL
ScL
c)

Fig. 5. A graphical interpretation of the relationship of TL, ScL and SpL: a) view of PROMOL as SpL + ScL; b) view of VHDL + PROMOL; c) VHDL in role of TL, SpL and ScL.

3) As a number of GC's is small and a number of instances produced from those GC's may be large, a *processor* that interprets those GC's can be treated as a *program generator* for a narrow application.
4) VHDL contains many advanced internal abstractions that allow developing a generic component. A generic component is a powerful mechanism for implementing reuse. The framework for the development of GC's is as follows. First, we need to formulate the requirements as precisely as possible. Next, we need to analyse the application domains for which we intend to use the developed GC. Finally, we must map the given set of requirements and user's needs expressing them with the abstractions that allow generalising a component.

5. The Extended Models and their Implementation in Experimental Systems

In this Section we will introduce several experimental generating systems based on the 3L paradigm. The cases (see Fig. 2, *b*) can be categorised (combined) into three categories as follows:

a) PROMOL processor as a stand-alone generator (case 1, case 3, case 4, case 5, case 6, case 8, case 9);
b) generator model that uses DSL (VHDL) abstractions only (case 2);
c) generating system that uses the PROMOL processor as a subsystem (case 7).

In Fig. 6 we deliver the extended representation of our model. This representation covers DA of domains under consideration, too. We restrict here the application building and analysis for a few domains. The examples are: VHDL-oriented applications, applications related with various types of transformations, such as a Chebyshev's approximation

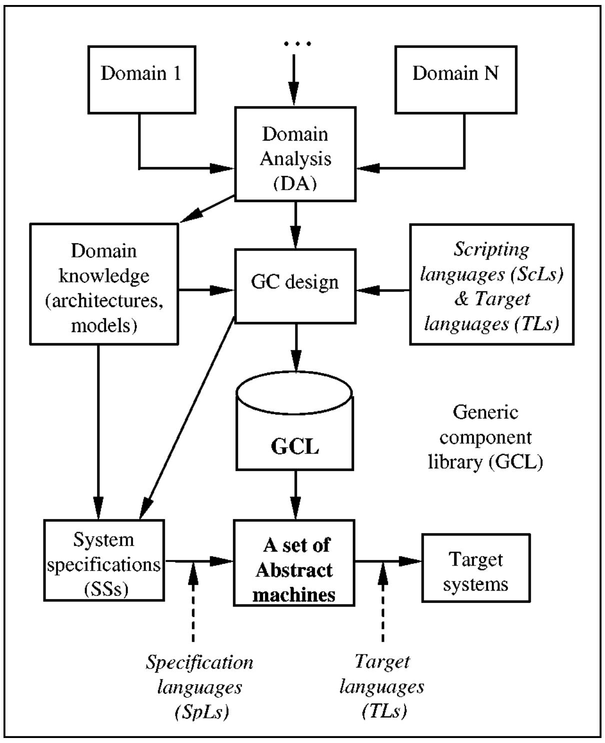

. . .
Domain 1
Domain N
Domain Analysis (DA)
Domain knowledge (architectures, models)
GC design
Scripting languages (ScLs) & Target languages (TLs)
GCL
Generic component library (GCL)
System specifications (SSs)
A set of Abstract machines
Target systems
Specification languages (SpLs)
Target languages (TLs)

Fig. 6. The extended view of the 3L paradigm.

of functions, Fast Fourier Transformation, etc. The latter applications are also known as data specialisation (Chirokoff *et al.*, 1999) or program specialisation (Glück *et al.,* 1995).

Other cases are described in our works (Štuikys, 2000; Ziberkas, 2000; Štuikys *et al.*, 2000). In the next subsection we present the role of the generic component library (GCL) in our approach.

5.1. *Role of GCL in the 3L Paradigm*

The library concept is fundamental for many tools and applications. It allows implementing the "black-box" reuse and achieving a higher productivity in software systems. The library model used in our generator's paradigm allows to combine two approaches, the compositional and generative reuse, and extract the inherent efficiency ("use-as-is", productivity) that resides in each approach. Our library model has the following characteristics.

First, the number of its items is relatively small because the items are generic components (GC's). We produce a GC using some abstraction that allows representing a family of related instances of domain objects, in many cases, with one component. Next, we allow different abstractions (either internal abstractions of the TL itself, such as generics, packages, unconstrained array types, etc., or external abstractions, such as external

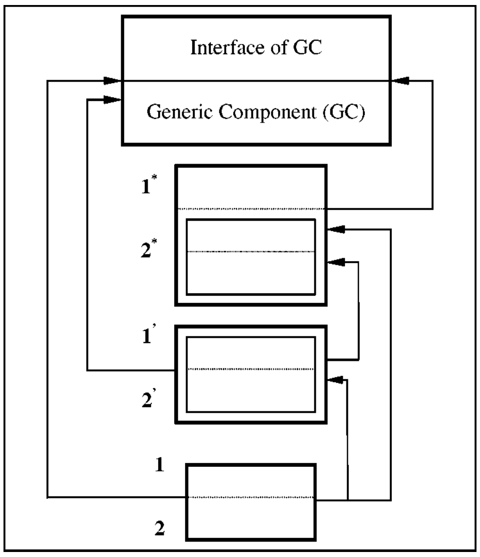


Fig. 7. GC model for a given application domain.

(1 – interface of VHDL component instance; 2 – VHDL implementation (or its functionality); 1′ – interface of GC developed using VHDL abstractions only; 2′ – GC implementation using VHDL abstractions only; 1– interface of GC developed using mixed abstractions (PROMOL + VHDL); 2*– GC implementation using mixed abstractions).*

functions of the higher-level scripting language, or mixed abstractions). This ensures flexibility for the system. Finally, the component structure is unique in our model no matter what abstraction we use (Fig. 7).

The structure, i.e., the library item, consists of two interrelated parts: the interface and implementation. In general, the interface serves for communicating with the environment or other components. As we have the hierarchical environment (system-level, processor-level), the interface has the hierarchical structure, too (see Fig. 7). Note that the lowest level interface of a component (1 or 1' in Fig. 7)serves for communicating with other components. The implementation serves for implementing a functionality or algorithm (generalised or concrete) of the component. This view to the component is common for each component type including the library item (it is represented at the highest level of the model). The interface of a library-level GC contains the following data: 1) the status of a component: TL, certification level (number of test cases, type of tools used for validation, etc.); 2) characteristics of a component: name, data (signal) types, abstraction used, characteristics of its functionality, application domain, statistics.

The structure of GC presents the component model. It is visible in the model (see Fig. 7) as a single structure, i.e., it is represented with respect to the highest level external agent (human or system). However, the internal structure of GC may be a complex system consisted of the lower-level components. In such a case we say that a GC represents an architecture of a domain application. Furthermore, GC as architecture does not depend

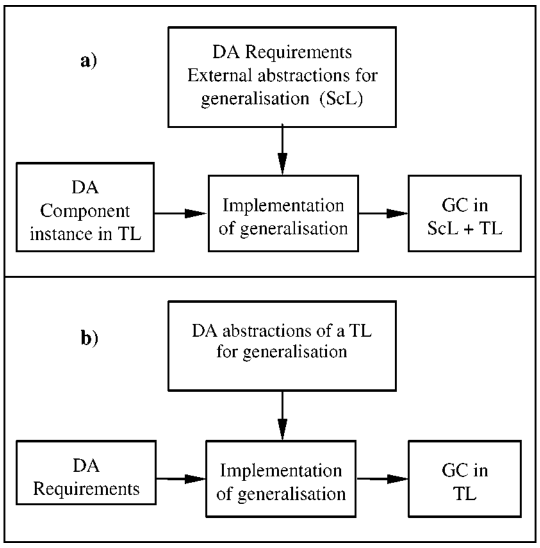

a)
DA Requirements External abstractions for generalisation (ScL)
DA Component instance in TL
Implementation of generalisation
GC in ScL + TL
b)
DA abstractions of a TL for generalisation
DA Requirements
Implementation of generalisation
GC in TL

Fig. 8. Two approaches for developing a GC: a) with external abstractions; b) with internal abstractions of a TL.

upon abstraction used for composing that GC (see Fig. 8).

The experiments reported in (Ziberkas, 2000) have shown that both abstractions (VHDL only and VHDL+PROMOL) are applicable and, in many cases, yield the very similar result of the synthesis. We don't have the intention to show a comprehensive comparison of these two abstractions here. Our recommendation is to apply both in GC design.

To build GCL for a generator we need, first, to undergo through analysis and, next, through the implementation of a set of requirements that may be categorised into two categories: 1) user requirements; 2) tool requirements. The first category mainly relates to (sub)domain objects, their characteristics and attributes, such as functionality and efficiency of GC's. For different domains the efficiency criteria may be different. For example, for the VHDL-related applications and given technology, the efficiency criterion is either the area of the chip that can be produced from the given component or the input-output delay. For transformation-based domains a more likely criterion will be the performance of a target program.

As for tool requirements, the most important requirements are as follows: domain boundaries and applicability (the number and quality of GC), compatibility of GC's, efficiency in use (adaptability, extensibility and maintainability).

5.2. *PROMOL Processor as a Stand-Alone Program Generator*

Suppose we possess the GCL containing items that have the structure as described in Fig. 8. Suppose we have a program that performs the following actions: a) reads an item from the GCL; b) recognises the type of abstractions used in the GC; c) transfers the GC to PROMOL processor if the GC possesses the PROMOL abstractions. Let us now consider

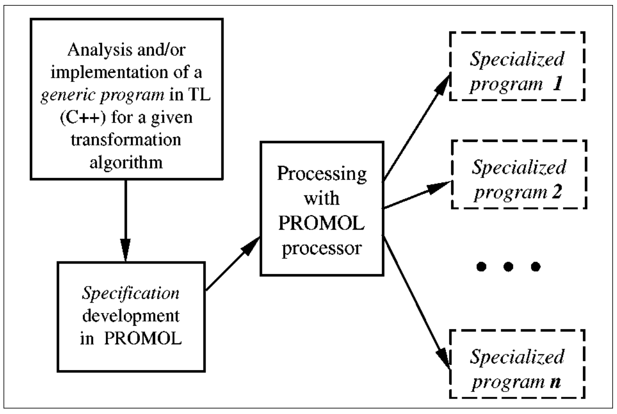

Analysis and/or implementation of a *generic program* in TL (C++) for a given transformation algorithm
Specification development in PROMOL
Processing with PROMOL processor
Specialized program 1
Specialized program 2
• • •
Specialized program n

Fig. 9. PROMOL processor as a stand-alone C++ code generator.

a system, which consists of GC and PROMOL processor. The latter interprets a GC. The way in which the interpretation follows depends upon the selected mode for the process. For example, a user can choose a mode in which the interface program of GC will be processed using the default parameter values (see Fig. 2 and 3). In this case the problem specification phase is omitted because the processor uses the *default specification*. If a user needs to change the problem specification program (aka interface program) he (she) should select another mode allowing to perform modifications of that program. As has been stated previously, we use the PROMOL processor for different applications. The tool stands for a program generator for those domains. How the PROMOL processor stands for C code generator, we will illustrate below.

Take, for example, the transformation-based applications (Glück *et al.*, 1995; Chirokoff *et al.,* 1999). The main intention of the PROMOL usage in such applications is to show the applicability of the approach for computing time *reduction* and increasing performance of the algorithms. This can be achieved assuming that the execution of a target program may be split into two phases: the *early phase* and *late phase*. The early phase implements the *static computing* of the transformation algorithm, while the late phase implements *dynamic or run time computing*. At the earlier phase using PROMOL we carry out calculus, which are time-consuming such as $cos(x)$ and also perform *enrolling* of the loops. As a result, the PROMOL processor creates a set of target programs called *specialised programs* each containing the different number of points. A generation of the specialised program follows as it is stated in Fig. 9.

The usage of the PROMOL processor for specialisation of C++ programs does not require the full potential of the language. We need a subset of the language to be used in this case. Furthermore, a specification in PROMOL developed for this application represents a stand-alone component. This specification may be treated as a *fine-grained component* no matter that the instances produced by this component may have a large size (this will become evident after analysing the experimental results).

We can demonstrate wider capabilities of the language when developing with PROMOL the *course-grained* but *stand-alone components* in VHDL. Let us return to the

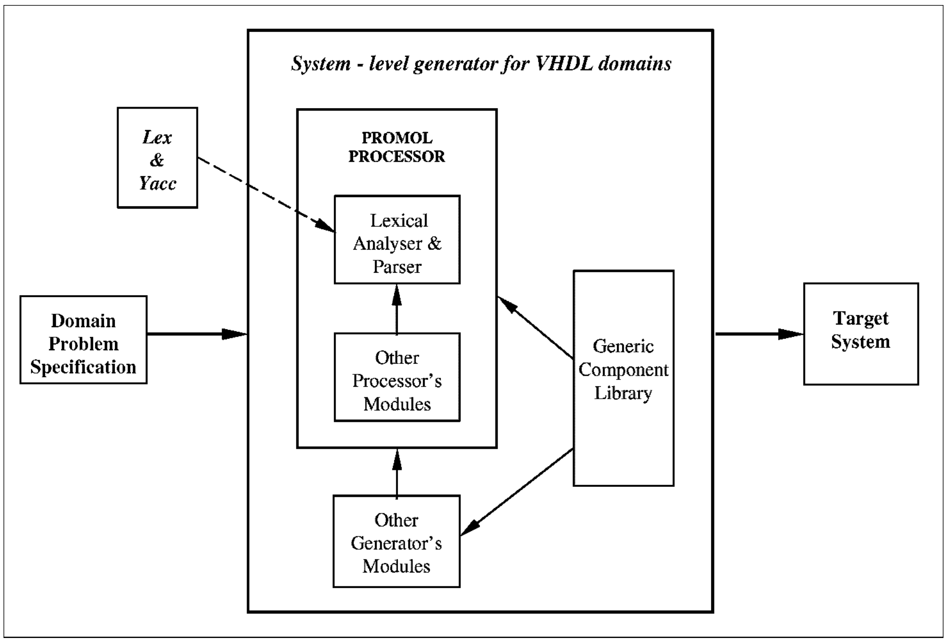

System - level generator for VHDL domains
Lex & Yacc
PROMOL PROCESSOR
Lexical Analyser & Parser
Domain Problem Specification
Other Processor's Modules
Generic Component Library
Target System
Other Generator's Modules

Fig. 10. A simplified architecture of the system-level VHDL generator.

model given in Fig. 7. This model allows to implement a more complicated system, actually an application in a given domain. However, this application should be represented as a stand-alone instance. The specification itself may consist of external modules as it is shown in (Štuikys, 2000). But this model can also be generalised in the following way. One can assume that a particular external module may be a constituent of several independent root modules.

5.3. *A Generating System Using the PROMOL Processor as a Subsystem*

Till now we have analysed generating systems that have had a single specification or, in other words, GC and its processor. As has been stated in previous sections, a GC itself may represent a system containing lower-level components. A great deal of instances can be produced through processing. No matter how large or small CG is, there is always a need for user to compose a system from components by linking components externally, not only using the hidden (prescribed) packaging of smaller components in GC. The external composition of a system is more flexible because user can implement the particular requirements.

The simplified architecture of the system-level generator is given in Fig. 10. As *Lex&Yacc* is an application generator, our system implements three-level generator's model:

- system-level generator (highest level),
- *promol* processor (middle level),

```
entity GATE_SYSTEM is
   port(d : in bit_vector(1 to 4);
          p : out bit);
end GATE_SYSTEM;

architecture PARITY_CHECK of GATE_SYSTEM is
signal s1, s2 : bit;
component GATE_XOR     port(x1, x2 : in bit; y : out bit);
end component;

begin
   u1 : GATE_XOR port map(x1=> d(1), x2=> d(2), y=> s1);
   u2 : GATE_XOR port map(x1=> d(3), x2=> d(4), y=> s2);
   u3 : GATE_XOR port map(x1=> s1, x2=> s2, y=> p );

end PARITY_CHECK;
```

Fig. 11. The parity control system composed from 3 XOR gates.

- *Lex &Yacc* (lowest level).

A user carries out the domain problem specification using graphical SpL. The output of the generating system is a target system described in VHDL. If a GC has only the VHDL abstraction, the PROMOL processing cycle is omitted.

Here we will formulate some problems, which may arise when implementing the system level generator:

1) domain analysis and GCL design;
2) compatibility of the components to make up a system;
3) integration of the lower-level subsystems into the generating system;
4) a specification problem for generating application for a given domain.

These problems are interrelated and should be considered more thoroughly. We consider briefly only the last mentioned problem in the next subsection.

5.4. *Graphical SpL Versus Textual SpL*

For domains related with hardware design the VHDL structural subset can play a role of the SpL (see Fig. 11). From the user's viewpoint, it is more convenient to use the graphical language, but not an abstract textual language. As for the usage of the structural subset of VHDL, we can observe some correspondence between the graphical and textual representation (see Fig. 11 and 12). Furthermore, with the textual representation in mind, we have a case when all three languages (TL, ScL, SpL) are different.

Regardless of some agreements (standards) for representing the domain objects such as gates and their attributes graphically, we need to have a higher accuracy for graphical representations with respect to the accuracy that has the textual representation. We

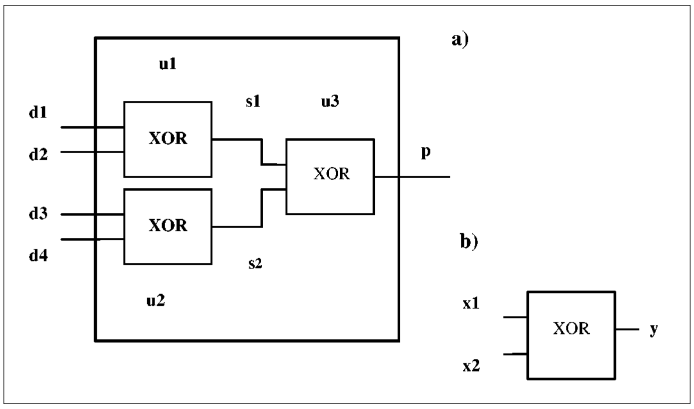


Fig. 12. A graphical representation of two-level parity control system (a) and component (b).

illustrate the difficulties that may arise for graphical representation with the following example. Suppose we need to deliver the generic delay constant between the input-output chain in the **XOR** gate in the above mentioned parity check system. It can be written in VHDL, for example, for gate **u1** as follows:

```
component GATE_XOR
  generic (T: TIME);
  port(x1, x2: in bit; y: out bit);
end component;
u1: GATE_XOR generic map (T => 2 ns)
  port map(x1=> d(1), x2=> d(2), y=> s1);
```

It is not quite clear how the same attributes should be presented graphically not loosing the representation accuracy and clarity. One of the solutions of the problem might be such as that implemented in the experimental generating system described in Section 5.2. In this system the both representation forms, the graphical and textual SpL, are used. A user specifies a domain problem (system) interactively using graphical icons for representing domain objects and lines for representing wires (signals), the nets for linking objects. Prompts are used for specifying particular attributes, such as generics, that must be incorporated into the description. These attributes are not the part of the graphical representation. They are incorporated in the internal system model only. At the system specification phase, the textual representation of the system to be created is hidden from user, too. A generating system creates the full textual representation in VHDL only after completion of the specification phase.

One can assume that this two-stage specification scheme is a compromise between two interrelated representation forms, the graphical and textual languages. Furthermore, we can consider that the specification created in the structural VHDL is a part of a target program given in the TL.

Table 1

Comparison of two approaches for program specialization of the FFT algorithm *(Glück et al. 1995)*

Program type	Number of points for FFT	Perform-ance in μs (our result)	Perform-ance ratio	Perform-ance in ms *(Glück et al. 1995)*	Perform-ance ratio *(Glück et al. 1995)*	Program volume in bytes (our result)	Program volume ratio	Program volume in lines *(Glück et al. 1995)*	Program volume ratio[1] *Glück et al. 1995)*
unspecialised	16	80.41	6.69	2.17	5.05	615	7.59	151	4.64
specialised		12.02		0.43		4667		701	
unspecialised	32	191.98	4.56	5.71	4.17	615	17.94	151	9.72
specialised		42.70		1.37		11035		1469	
unspecialised	64	514.65	3.10	15.27	3.56	615	41.38	151	20.32
specialised		166.00		4.29		24446		3069	
unspecialised	128	1245.80	2.91	40.86	2.86	615	94.90	151	42.36
specialised		427.60		14.22		58363		6397	

[1]**Note**. Other implementation language and different measurement units used, perhaps, can explain different ratio of program volume.

6. Experimental Results Gained from the Systems

We deliver the partial results (see Table 1 and 2) gained from the experimental system. The primary objective of the experiments is to advocate the applicability of the approach.

Table 1 illustrates the applicability of PROMOL processor as a stand-alone program generator for specialisation of a target program in C. Actually we regenerated with our approach the results given in (Glück *et al.*, 1995). Non-adequate environments in the experiments can explain small discrepancies in the program speedup and memory losses. Note that we have performed the program evaluation (analysis) in that experiment manually and automated the specialised program generation only.

Table 2 delivers an illustrative example for comparison of three different generating models for VHDL applications. No matter that each model has a slightly different source code (syntactically but not semantically) and it has been created in different way, the final solution (the results of the synthesis) is very close for the same domain object (system). The discrepancies are due to the inadequate interpretation of the source code by synthesis tools we have used for experiments.

7. Summary and Evaluation of the Results

Our approach for building generators can be summarised as follows.

Firstly, we create a generic component library (GCL) for a given application domain. A generic component (GC) is a generalised specification representing the different functionality (features) of the same domain object. A GC represents a set of component instances which, when applied, yields a concrete instance derived from that GC. As we use GC's, the size of the library is usually small. Our generators work in two modes: 1) a generator is a system that produces the stand-alone instances in a given target language from

Table 2

Comparison of the results gained from three generation models for VHDL domain

Domain object (system)	List of components	VHDL code size in KB		Chip characterictics			Architecture size in bits
		Generic	Instance	Number of cells	Area	Delay in ns	
Adder created from GA[1] (VHDL abstraction only)	GG1[2], 1 bit Adder	4.9	4.9	17	10,320.8	3.08	4
		4.9	4.9	37	20,748.0	5.65	8
		4.9	4.9	77	41,602.4	10.81	16
		4.9	4.9	157	83,311.2	21.18	32
Adder created from GA (VHDL+PROMOL)	GG2[3], generic gate system equations	3.2	2.6	19	10,366.4	3.18	4
		3.2	2.8	39	20,793.6	5.95	8
		3.2	3.3	79	41,648.0	11.50	16
		3.2	4.4	159	83,356.8	22.66	32
Adder generated by system-level generator from GG3[4]	Gate instances derived from GG3	0.3	6.1	12	10,244.8	3.33	4
		0.3	9.5	24	20,489.6	5.60	8
		0.5	8.5	48	40,979.2	10.47	16

[1]GA – generic adder.
[2]GG1 – generic gate composed with VHDL abstraction only.
[3]GG2 – generic gate composed with VHDL+PROMOL abstractions.
[4]GC3 – other generic gate composed with VHDL + PROMOL abstractions, Cadence Synopsis synthesis tools have been used.

GC's (VHDL instances, C specialised programs, etc.); 2) a generator produces a VHDL system (application) composed from VHDL instances while the instances are generated from the GC's.

Secondly, and this is the most important, we introduce the following abstractions: a) the target language abstractions to express different variations of a domain functionality; b) the external language abstractions (the PROMOL abstractions); c) the mixed abstractions. More specifically, these abstractions allow us to build the relationship model for representing domain functionality, scripting variations of a different functionality and specifying a domain problem. More generally, we present this model as a paradigm that describes the relationship of three languages (target, scripting and specification) for implementing generation systems.

Finally, at the implementation stage, we use other tools as "components-from-the-shelf": the *Lex&Yacc* for implementing the PROMOL processor, and the *PROMOL processor* for implementing generators. The system-level VHDL generator, for example, explores the VHDL capabilities for specifying the target system to be built. However, we use the graphical interface for specification of a target system. This allows hiding the language abstractions from the user. The GCL and PROMOL processor are non-visible for the user, too. We analyse several domains and carry out experiments with experimental systems.

From the conceptual standpoint, other authors also use hybrid approaches, which combine pure components with some generation technology (Biggerstaff, 1998; Batory *et al.*, 1995; Batory, 1998b). The systems reported in the literature, perhaps, have a higher maturity[2] level. However, we have achieved the presented results in a specific way demonstrating the applicability and domain independence of the approach.

8. Concluding Remarks

VHDL, as a domain-specific language, allows expressing the domain content well and implementing this content in generating systems. The generative factors predominate over other technology factors. Our approach combines the following technological factors: the domain-specific and conventional languages, componentry and generative factors. However, for purposes of generalisation, i.e., generic component building, and higher flexibility for expressing different variations of a domain functionality, we introduce the external language, called Open PROMOL, as a complementary to the given target language. The external language allows enhancing the specification capabilities of a generating system, too.

We have developed several experimental systems, which have approved the 3L paradigm. We have achieved this result in a specific way. We demonstrate the following result: a) the role of domain content and domain-specific language to express this content; b) the independence of Open PROMOL from a given domain and a target language; c) the role of the mixed abstractions for implementing generation systems; d) the applicability of the approach for building generating systems.

Acknowledgements

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

References

ARINC Specification 608–1 (1989). Standard modular AVIONICS Repair and Test System SMART. September.

Ashenden, P.J. (1996). *The Designer's Guide to VHDL*. Morgan Kaufman Publishers, Inc., San Francisco, California.

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

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

Batory, D. (1998a). Product-line architectures. *Invited Presentation, Smalltalk and Java in Industry and Practical Training*. Erfurt, Germany, October, 1–12.

Batory, D. (1998b). Domain analysis for GenVoca generators. *IEEE Software*, September, 350–351.

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

Biggerstaff, T. (1998). A perspective of generative reuse. *Annals of Software Engineering*, **5**, 169–226.

[2]Authors (Batory, Biggerstaff) have reported that their (or analyzed) results are used in industry, while our method is used in experimental systems.

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

Chirokoff, S., C. Consel, R. Marlet (1999). *Combining Program and Data Specialization. Higher-Order and Symbolic Computation 12*. Kluwer Academic Publishers, Netherlands, 309–335.

Czarnecki, K., U. Eisenecker, R. Glück, D. Vandervoorde, T. Veldhuizen (1999). Generative programming and active libraries (Extended abstract).
`http://extreme.indiana.edu/tveldhui/papers/dagstuhl1998/dagstuhl.html.`

Glück, R., R. Nakashige, R. Zöchling (1995). Binding-time analysis applied to mathematical algorithms. In J. Dolevar, J. Fidler (Eds.), *System Modelling and Optimization*. Chaman& Hall, pp. 137–146.

Griss, M. (1993). Software reuse: from library to factory. *IBM Computer Journal*, **32**(4), 548–566.

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

IEEE Standard ATLAS Test Language (1981). IEEE std. 416.

Iscoe, N., G.B. Williams, G. Arango (1991). Domain modelling for software engineering. In *13th International Conference on Software Engineering*, Austin, Texas, USA, May 13–16, pp. 340–343.

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

Johnson, S.C. (1975). Yacc – yet another compiler. *Computing Science Technical Report*, **32**. AT&T Bell Laboratories, Murray Hill, N.J.

Lesk, M.E. (1975). Lex – a lexical analyzer generator. *Computing Science Technical Report*, **39**. AT&T Bell Laboratories, Murray Hill, N.J.

Levine, J.R., T. Mason, D. Brown (1992). *Lex and Yacc*. O'Reilly and Associates, Inc., Sebastopol, CA.

Lim, W.C. (1998). *Managing Software Reuse.* Prentice Hall PTR.

Luker, P. A., J. Stephenson (1979). Program generation for continuous system simulation. In Dekler (Ed.), *Simulation of Systems*. Amsterdam, North-Holland.

Luker, P.A., A. Burns (1986). Program generators and generation software. *The Computer Journal*, **29(4)**, 315–321.

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

Milne, G. (1994). *Formal Specification and Verification of Digital Systems.* McGraw-hill Book Company, London.

Neighbors, J.M. (1998). Domain analysis and generative implementation. In *IEEE Symposium on Software Reuse*, pp. 356–357.

Oswald, B.S., J.S. Greg (1983). Automatic ATLAS program generator (AAPG) for the advanced electronic warfare test set. In *AUTOTESTCON'83*, pp. 286–291.

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

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

Phelps, R. (1987). New tools automate the application development cycle. *Hardcopy*, January, 138–141.

Prieto-Diaz, R. (1990). Domain analysis for reusability. In W. Tracz (Ed.), *Software Reuse: Emerging Technology*. IEEE press, pp. 347–353.

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

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

Štuikys, V., E. Toldin (1993). Computer-aided test program design system (CATPDS) in ATLAS. *Informatica*, **4**(4), 389–405.

Štuikys, V., G. Ziberkas (1997). An approach for building VHDL code from domain problem reusable templates. *Information Technology & Control*, **3**(6), 46–55.

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

Štuikys, V., R. Damaševičius (2000). Scripting language open PROMOL and its processor. *Informatica*, **11**(1), 71–86.

Štuikys, V. (2000). The development of specifications in Open PROMOL for modifying the target language programs. *Information Technology & Control*, **1**(14), 30–45.

Teitelbaum, T., T. Reps (1981). The Cornell program synthesizer: a syntax-directed programming environment. *Communication of the ACM*, **24**(9), 563–573.

Terry, Ch. (1987). CASE tools runs on an expanded range of computer systems. *Electronic Design News*, **23**, 221–228.

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

Thibault, S., C. Consel (1997). A framework for application generators design. In *Symposium on Software Reuse'97*. ACM, pp. 131–135.

Thibault, S., R. Marlet, Ch. Consel (1999). Domain-specific languages: from design to implementation application to video drivers generation. *IEEE Transactions on Software Engineering*, **25**(3), 363–377.

Tinaztepe, C., N.S. Prywers (1979). Generation of software for computer controlled test equipment for testing analogue circuits. In *IEEE Transactions on Circuits and Systems*, Vol. CAS-26, No. 7. pp. 537–550.

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

Ziberkas, G. (2000). The development of generic components in VHDL. *Information Technology & Control*, **2**(15), 51–58.

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.

G. Ziberkas received a bachelor degree in informatics from Kaunas University of Technology in 1995 and MSc degree in 1997. He is currently Ph.D. student at Informatics faculty, Kaunas University of Technology. His research interests include hardware design with VHDL, software design, and software reuse.

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.

Trijų kalbų paradigma programų generatorių modeliams kurti

Vytautas ŠTUIKYS, Giedrius ZIBERKAS, Robertas DAMAŠEVIČIUS

Straipnyje pasiūlyta paradigma programų generatorių modeliams kurti. Paradigmos esmę sudaro 3 kalbos: tikslinė, scenarijų (klijavimo, komponavimo) ir specifikavimo. Mes parodome, kad net viena į problemą orientuota kalba gali atlikti šių trijų kalbų vaidmenį. Paprastai srities architektūrą, t.y. generinius komponentus mes aprašome vartodami dvi kalbas (tikslinę ir scenarijų). Mes pateikiame šių trijų kalbų roles, santykį (ryšius) ir ribas iš generavimo sistemos sukūrimo pozicijų duotajai sričiai. Valdymo ir transformavimo galimybių atžvilgiu mūsų modeliai gali būti traktuojami kaip abstrakčios mašinos arba apibendrinti transliatoriai aukštesnio lygio programoms kurti. Mes pateikiame kai kuriuos eksperimentus pasiūlytam metodui validuoti.