INFORMATICA, 2002, Vol. 13, No. 1, 111–128
© 2002 *Institute of Mathematics and Informatics, Vilnius*

Relationship Model of Abstractions Used for Developing Domain Generators

Vytautas ŠTUIKYS, Robertas DAMAŠEVIČIUS
Software Engineering Department, Kaunas University of Technology
Studentų 50, 3031 Kaunas, Lithuania
e-mail: vytautas.stuikys@if.ktu.lt

Received: June 2001

Abstract. In this paper, we analyze the abstractions used for developing component-based domain generators. These include programming paradigms, programming languages, component models, and generator architecture models. On the basis of the analysis, we present a unified relationship model between the domain content, technological factors (structuring, composition, and generalization), and domain architecture. We argue that this model is manifested in the known software generator models, too.

Key words: software generation, domain architecture, multi-paradigm design, model generator.

1. Introduction

Software components are building blocks to make up a complex system. The composition of a system from components is a fundamental issue of any design. The component-based approach serves for achieving better quality and higher productivity. It is widely believed that reusability is a key for improving productivity and quality. In this context, Szypersky evaluates the component-based approach as the "*law of nature*" that resides in any mature engineering discipline (Szypersky, 1998). The component-based and generative approaches are two main directions in "technological" reuse. In recent years these directions became much closer to each other than they were, say, ten years ago. The explanation is simple – the researchers and practitioners have clearly understood that software generation should be built on the sound and proven model, i.e., on the concept of a *library component*. The increasing stream of publications continues to appear on these topics. There are not only the interesting generator models but their implementations in industry settings, too. The known contributions differ from each other in many technological aspects, however, they have as well many in common from the conceptual point of view.

The aim of this paper is to analyze the generative approach and different abstractions most frequently used (templates, generics, meta-programming, scripting, etc.) with the intention to build a unified "relationship model" between the abstractions and domain content (domain architecture) when implementing generic components and generators. We hope that such a model might (1) contribute to the better understanding of existing generator models; (2) clarify the process of designing software generators; and (3) encourage further research in this area.

The structure of the paper is as follows. In Section 2, we present the backgrounds for the analysis. In Section 3, we analyze the abstractions including programming paradigms, languages and architecture models. In Section 4, we discuss the basis of the proposed model. In Section 5, we deliver the relationship model between domain content, technological factors and domain architecture. The evaluation of the generator models is presented in Section 6. Finally, in Section 7, we present the discussion and conclusions.

2. The Background for the Analysis

The development of complex systems, such as software generators, is a major problem in software engineering. The usual way to managing complexity is to apply the principle of *separation of concerns*. Although Dijkstra has coined the concept in the 1960's, up to recently it was used mostly at the component level, i.e., to reduce the complexity of algorithms by dividing the problem into several smaller ones, and implementing them separately. The most evident result of employing this "*multi-component design*" strategy is the usage of component libraries. However, since these times the complexity of developed systems has grown significantly, thus the complexity problem should to be dealt with at a higher level of abstraction.

The authors (Coste *et al.,* 1999; Jerraya *et al.*, 1999) have proposed a concept of *multi-language design*. They argue that the usage of the different languages that are more suitable and efficient for the specification of the subsystem of a large system is inevitable when the system has to be designed by separate groups that are using different design methods, languages and tools. The multi-language specification of a system implements the principle of separation of concerns at the level of programming languages. The advantage of using the domain-specific language (DSL) that is oriented at the solution of a particular sub-problem is evident. However, not all problems can be addressed and solved at the language level. Some problems (e.g., generalization) can be effectively dealt with only if a different programming paradigm was employed.

The authors (Horspool *et al.,* 1993; Spinellis, 1994; Coplien, 2000) consider the concept of *multi-paradigm design*. It supports the software development based on several design paradigms simultaneously. The multi-paradigm design is not only focused on the problem domain, but on the solution domain, too. The goal of the analysis of the solution domain is to find software abstractions and programming technologies that naturally express the domain knowledge gained by analyzing the problem domain. The result of such an analysis is the model of the designed system that bridges over the different component models, programming languages and programming paradigms. We accept the principles of the multi-paradigm design as a background for analysis we present below.

3. Review and Analysis of Generative Abstractions and Models

3.1. *Programming Paradigms*

We argue that three distinct programming paradigms could be used for implementing generators and their components. These are: (1) *algorithmic* or *object-oriented programming* – components are implemented as structures (functions, procedures, objects) which *encapsulate* specific code fragments; (2) *meta-programming* – components are implemented as meta-programs, which *generate* specific component instances; and (3) *scripting* – components are implemented as black-box entities, which are *glued* together using a scripting language. The paradigms may be used separately or combined together when implementing generators. Below we present the overview of those paradigms.

Object-oriented programming (OOP) has yielded a major advance in software engineering. Its main concepts (encapsulation, inheritance, polymorphism) have contributed greatly to the extensibility, adaptability and reusability of components (the role of OOP for reuse has been evaluated in (Edwards, 1999)). The main advantages of OOP can be summarized as follows: (1) *encapsulation* supports an abstraction and provides better management of complexity; (2) *inheritance* allows extending and modifying objects; and (3) *polymorphism* provides means for an alternative implementation of components.

Additionally, a great number of *design* patterns (Gamma *et al.*, 1995) were established, which (1) help to reuse patterns of interaction between objects; and (2) support the creation of OO frameworks. Design patterns provide reusable solutions for families of applications instead of developing single applications from scratch, and are widely used in a number of generators (e.g., see (Budinsky *et al.*, 1996)), especially in the graphical user interface (GUI) generators.

However, OOP does not provide adequate means for *generalization* in the broad sense. According to Sametinger (Sametinger, 1997), OOP supports generalization by *narrowing* only. Other generalization forms, such as *widening,* can be implemented using *meta-programming* techniques. OOP does not address the issue of *composition* adequately, too. According to Nierstrasz and Meijler (Nierstrasz *et al.,* 1995) OOP (1) fails to clearly distinguish the computational and compositional views of applications; and (2) over-emphasizes the object view, thus failing to provide general component specification and composition mechanisms. These problems can be solved using the *scripting* technology.

Meta-programming is a programming technique that enables a manipulation with other program structures. To distinguish between different program layers, a lower layer of abstraction is denoted as a *target language* (TL), and a higher layer of abstraction – as a *meta-language*. In linguistics, a meta-language is defined as follows: *"any language or symbolic system used to discuss, describe, or analyze another language or symbolic system"* (Czarnecki *et al.*, 1998). A program that manipulates another program is clearly an instance of meta-programming, and is called a *meta-program*. According to Batory (Batory, 1998), a meta-program is "*a program that generates the source of the application ... by composing pre-written code fragments*". In other words, a *meta-program* is a set of instructions, descriptions, and means of control (possibly *generation*) of sets of target programs. Meta-programs may be written using the same programming principles and constructs (*if, case, for loop*) as target programs, however, they manipulate on *program* representations, not on *data*.

As separate cases of meta-programming can be considered: (1) Generative Programming; and (2) Aspect-Oriented Programming. Further, we consider them in some detail.

Generative Programming (GP) is a program development approach for generating customized components and systems (Eisenecker, 1997; Czarnecki *et al.,* 2000). GP is based on the central themes of domain engineering and meta-programming. The focus is on automating the mapping between the *problem* and *solution* domains given by an established *architecture*. GP aims to (1) decrease the conceptual gap between domain concepts and program code; (2) to achieve higher component reusability and adaptability; (3) to simplify managing of component variants; and (4) to increase efficiency.

To achieve the prescribed aims GP uses the following techniques: *parameterization, separation of concerns, DSL techniques,* and *configuration knowledge*. The parameterization increases reusability by providing parameterized components, which can be instantiated for different choices of parameters. The principle of separation of concerns separates each domain problem into a distinct generic component or sets of components used to generate target program. The DSL techniques allow achieving domain-specific optimizations and better code understandability. The configuration knowledge allows capturing a specific information about the parameter dependencies, default settings and illegal combinations. The concepts of GP are implemented in *Active Libraries*, which extend conventional component libraries by providing (1) domain-specific abstractions with automatic means of producing optimized program code; and (2) debugging, profiling and testing capabilities.

Aspect-Oriented Programming (AOP) (Kiczales *et al*., 1997) is a way of modularizing features that cut across the traditional modular partitions. The common considerations like performance and reliability usually manifest themselves in code fragments scattered about a system. AOP decomposes domain problems into *components* and *aspects*. The structure of the AOP-based specification of an application is a typical example of a multi-language specification. A component is understood as a generalized procedure (i.e., function or object) implemented using the *component* (i.e., *target*) *language*. An aspect is a property of the component that affects its performance or semantics in a systematical way, and is implemented using the *aspect* (i.e., *meta-*) $language(s)$. Components and aspects are *woven* (i.e., joined) together (using a compiler or pre-processor) to obtain a system implementation that contains *tangled* code, i.e., mixed code from components and aspects. AOP is still in the early stages of development now, and is more focused on conceptual issues. Authors estimate themselves that AOP is 20 years behind OOP.

The main idea of the ***scripting technology*** is that an application developer only has to write a small amount of *wiring code* in order to establish a connection between components. It can take various forms, depending on the nature and granularity of the components, the nature and framework of the problem domain, and the composition model.

Ousterhout (1998) claims that scripting languages represent a different style of programming than system programming languages. They are intended neither for writing applications from scratch nor for implementing complex algorithms or data structures. Instead, the scripting languages assume a collection of existing components and are intended for plugging these components together. Therefore, they are sometimes referred to as *glue languages* or *system integration language*s. Ousterhout defines the term scripting language as follows: "*Scripting languages are designed for gluing applications. They provide a higher level of programming than assembly or system programming languages, much weaker typing than system programming languages, and an interpreted development environment*".

According to (Schneider *et al.*, 1999), scripting can be considered as a *higher-level binding technology* for component-based systems, which denotes abstractions for connecting components. The c*omponents* implement the provided functionality behind a standard interface, and generally represent the *stable* parts of applications, whereas s*cripts* plug components together, and represent the *variable* parts of applications. Scripting has proven very successful in rapid development of GUIs and in component frameworks, i.e., areas closely related to software generators. Next, we consider programming languages, which implement the concepts of programming paradigms in practice.

3.2. *Languages Used for Developing Program Generators*

A high-level language is the main abstraction for expressing the domain content and coding components and generators. Some languages (e.g., Ada, VHDL, C++) provide a meta-programming mechanism, called *generics* (or *templates,* in case of C++), for writing parameterized components (Musser *et al.,* 1994). We analyze and compare the VHDL (IEEE Std. 1076, 1996) and C++ capabilities for expressing composition and generalization of components below.

VHDL provides *generics* for writing parameterized models. The *generic* component specification represents a set of possible component implementations or instantiations. A particular instance is generated when instance-specific data is passed into a generic component. The use of generics for parameterisation of a structure and behaviour is essential for design reuse-applications, and helps to create reusable design blocks. VHDL provides the *generate* statement for generating *repetitive structures*. These are the concurrent VHDL constructs that may contain further concurrent statements for replication, and help to effectively produce iterative structures of a design. The *repetitive* form of the generate statement specifies a discrete range of values, and for each of these it generates an instance of the statements in the body of the generate statement. The *conditional* form includes a Boolean expression, which governs whether the statements comprising the body of that *generate* statement are included. The generics provide a level of abstraction that allows isolating many implementation details when developing a software system as well as reduce the risk of introducing errors when components are reused. Additionally, VHDL supports the structural composition capabilities to compose a system from components (the *port map* statement). These features of VHDL allow to reuse hardware designs (Meiyappan *et al.,* 1999) and build circuit generators for adders, multipliers, DSP functions, decoders etc., see (Stohmann *et al.,* 1996; Stohmann *et al.,* 1997; Zimmermann, 1997).

We consider the C++ capabilities for generalization and composition from the viewpoint of meta-programming as they are presented in (Veldhuizen, 1995). Templates were designed to support *generic programming*, but unintentionally provided the ability to write code generators and perform static computations. The template specifies only a generic skeleton for a class (or function) declaration. To complete the declaration, a programmer must supply a concrete value for each of the template's parameters. These parameters must be known at compile time. This causes the template to be *instantiated*: replacing all occurrences of the parameter with its value creates an instance of the template. The composition is performed using the syntactic expansion. C++ templates resemble a two-level language, where a "*meta-programming sub-language"* (i.e., templates) manipulates the base code. Control structures (e.g., *if/else, for*) can be realized in templates using a *template recursion* as a looping construct, and a class *template specialization* as a conditional construct. However, the syntax for writing such structures is very clumsy. The technique of template meta-programming has proven useful in template libraries (e.g., STL, MTL, Blitz++, see (Veldhuizen, 1995), which provide optimized linear operations for small, fixed-size vectors and matrices, as well as system C++ for hardware design).

We conclude that both languages (VHDL and C++) have higher- and lower-level structures that allow expressing a generalization when designing components. In general, these structures can be treated as the meta-programming ones. VHDL has capabilities for structural composition, too. No matter how strongly the standard languages support meta-programming, one-language paradigm is not flexible enough. Next, we analyze the evolution of the meta-programming concept and the implementation of the multi-language paradigm in known generator models.

3.3. *Implementation of Multi-Language Paradigm in Known Generator Models*

We consider Bassett's *frame technology*, Batory's *GenVoca* model and our approach below.

Frame technology (Bassett, 1997) is a generative technique for adapting the target program source code using a pure lexical manipulation. It fragments the programs into generic components called *frames*. A *frame* is a text written in any target language. Frames are organized into a *frame hierarchy* and form a generic architecture, from which programs incorporating specific variants can be produced. The topmost frame in a frame hierarchy, called a *specification frame*, specifies how to adapt the rest of the frame hierarchy to the requirements of a given variant. Frames can be adapted to meet requirements of a specific system by modifying frame code at *breakpoints*. Each frame can reference lower level frames in the hierarchy, and also inherit default behavior and parameters from higher level frames. Each frame contains either *frame commands* or *generic text*, i.e., text that may contain embedded parameters. The frame commands, in combination, can add, modify, delete, instantiate, select, and iterate any details of sub-frames down to individual symbols. A frame processor customizes a frame hierarchy according to directives written in the specification frame and assembles the customized system. The adaptable frames enable most complexity to be hidden, making sophisticated systems easier to design and easier to adapt to changing needs. Despite their simplicity, frames have been used with a great success to create programs of the significant size in the commercial information systems design.

The *GenVoca* model (Batory, 1997; Batory *et al.,* 1997) is a component-based reuse strategy in which the components play the role of *layers of abstraction* in the program construction process. Each layer encapsulates a pure abstraction or domain feature. The assembly of layers causes the generation of large-grained components, which are then further composed into the target application. The layer (called *realm*) is an abstract domain entity that exposes a standard interface and allows many different implementations, called *components*. In GenVoca, a *component* is the basic unit of software system and construction. It is a suite of interrelated variables, functions and classes that work together as a unit to implement a particular feature of a software system for a given problem domain. A *realm* is a library of plug-compatible components. A realm is defined by a standardized interface that consists of functions and classes. All components of a realm inherit this interface and may specialize it by adding data and function members to existing classes, and by adding new variables, functions and classes. Realms and components are implemented using special language P++, which is an extension of C++, and offers syntactic constructs for component *abstraction, encapsulation, parameterization and inheritance*. The GenVoca model uses two parameterization types: the *horizontal* one, where realms can be parameterized with other realms, data types and constants; and the *vertical* one, which specifies the components in lower layers. The horizontal parameterization allows the *instantiations* of components, whereas the vertical parameterization allows the *composition* of components. The composition is performed by the syntactic expansion of the parameterized components. A P++ preprocessor assembles target systems from the specifications of components and compositions of layers.

Our approach uses both the one- and multi-language paradigms for building generic components and generators (Štuikys *et al.,* 2001a). By one-language paradigm we mean the exploration and usage of the VHDL capabilities only. By multi-language paradigm we mean the usage of two independent languages simultaneously, i.e., VHDL as a TL and Open PROMOL as a scripting language (SL). We use the TL for transferring domain content and the SL for achieving generalization. By introducing the SL, we seek the following aims: (1) To gain higher flexibility in VHDL-based generic component designs. (2) To support the design platform independent from a hardware design language (Štuikys *et al.,* 2000a, Štuikys *et al.,* 2001b). And (3) to extend the generative approach with program transformations, such as program specialization (Štuikys *et al.,* 2000b).

We categorize components either as *instances* or *generic components* (GCs). An instance implements the concrete functionality, whereas a GC is a generalization of "similar" instances, and implements a variety of the related functionality. The generalization is important for several reasons as stated below. We can reduce the size of reuse libraries significantly, because a GC may represent a family of instances. The generalization increases reusability of the component significantly because a GC may be customized in the different contexts of its usage. The instances can be either instantiated or generated from the GCs automatically. Finally, for implementing system generators, the Intellectual Property providers or tool designers can use a reuse library containing the GCs.

We use the three-layered GC models based on the monolithic (M-), hierarchical (H-) and generative (G-) architectures (Štuikys *et al.,* 2001a). The M-architecture mostly deals with generalization of the fine-grained stand-alone components, H-architecture – with composition of the coarse-grained components, and G-architecture – with automatic generation of large-scale systems composed of M- & H-architecture GCs. What is important to note is that we use the models in both paradigms (i.e., the one- and multi-language ones).

Next, we discuss the basis for the relationship model between the domain content, programming abstractions and generator architectures.

4. The Background for the Relationship Model

Software generation is a process of creation of a target system from a high-level specification (Thibauld *et al.,* 1997). Both, a high level specification and a target system express the same domain content, though in a different form and manner. An act of generation does not change the content of a target system, because it already *potentially* exists in a system specification. The process of generation only *actualizes* a system by changing the *form* of its content. In fact, a software generation is only a partial case of software transformation and a transformation as a whole.

The problem of *transformation* is a long-standing issue in the history of science. Aristotle noticed two types of transformation in his *Metaphysics*: (1) *accidental* (e.g., water turns into ice; compare to "a program is compiled to an executable code"); and (2) *substantial* (e.g., wood burns to coal; compare to "a program in a hardware description language is synthesized to produce a chip"). In each case, there is a passive substratum, a *matter* (*hyle*), which is a constant basic foundation of every changing substance. However, a *matter* is only an abstract representation of all things. To become an object, it needs to be *formed*, i.e., to take a concrete *form* (*morphé*), which reveals the particular features of a *matter* pertaining to that object.

A *program transformation* is a meaning preserving mapping defined on a programming language (Paige, 1997). It is used for the derivation of programs from the formal specifications or old program versions in a *semantics preserving way*. What changes during the program transformation is the *syntax* of a program, but its semantics remains the same. In other words, a program transformation is a transformation defined on a programming language domain. The *semantics* of a language is a *matter* of program transformation, and the *syntax* is a *form* of it. Program transformation tools, such as compilers or processors, work on the same principles as software generators, though the conceptual gap between input and output they bridge is much narrower than in generators.

Returning to software generators, Aristotle's *matter* can easily be recognized in a *domain*. Indeed, a domain contains abstractly all physical phenomena and concepts that pertain to it, without giving the concrete details of implementation (e.g., circuit design domain contains a concept of *gate*, whose behavior can be described by Boolean functions). A domain content takes a particular form when some *technology* is introduced. This technology can be understood as *hardware* (e.g., chip) or *software* (e.g., program) that implements a domain entity and evidently manifests its structure and behavior (in this paper we discuss the software technology only). However, the domain content and software technology are not enough to build a domain generator, as well as a *matter* and a *form* are not enough to make an object. What are putting a *matter* and a *form* together are the *rules of formation* or *organization* (aka *in-formation*), i.e., the principles of applying a *form* to a *matter*, the most common representation of which are the laws of nature. Accordingly, a software generator needs an *organizational medium*, which determines "the rules of engagement" for the technological factors. We call this medium the *domain architecture*.

Before any system could be build, first, the domain must be analyzed (we do not consider Domain Analysis (DA) here). The result of DA is *domain knowledge*. This domain knowledge is always limited and only partially corresponds to the domain content. This is due to the limitations of human understanding, imperfection of DA methods, and limited time of analysis. Furthermore, this knowledge is not organized and, therefore, can not be used "as is". In other words, domain knowledge can be used only if it becomes *domain information*. The knowledge must be *formatted* (i.e., classified, organized) in order it to become a domain information. This is achieved by applying the *rules of organization*. The gaining of information is the continuous process: the more domain information is extracted from the domain content and recognized, the better we comprehend the domain itself, and vice versa. Domain information is the hierarchically organized set of the interrelated domain concepts. We consider *domain architecture* as an expression of the domain information in terms of software engineering.

Domain architecture is the heart of a domain generator. It describes and implements a complete set of actions, which lead from a specification of a domain problem to an implementation of a target system. Perry and Wolf (Perry *et al.,* 1992) define a software architecture as a combination of *elements*, *form* and *rationale*: '*'A software architecture is a set of architectural (design) elements that have a particular form. Properties constrain the choice of architectural elements, whereas rationale captures the motivation for the choice of elements and form*". We argue that generator architecture is a particular combination (actually synthesis) of *domain content* ("design elements"), underlying *software technology* ("form") and *domain architecture* ("rationale") that maps domain content onto software.

Summarizing, we present the different viewpoints to program generators below:

- a generator implements "*the product-line architectures*" (Batory, 1998);
- a generator *"creates derived components and various relationships from languages and templates*" (Jacobson *et al.*, 1997);
- 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*" (Lim, 1998);

- generators are "*compilers for domain-specific languages*" (Villarreal *et al.,* 1997; Deursen *et al.*, 1999).

This spectrum of opinions is another reason for seeking to find a *unified relationship model* between these concepts we present in Section 5.

5. The Relationship Model

The main idea for implementing reuse is that the *domain content* and *technological factors* should be considered as interrelated concepts. Biggerstaff (1998) describes this relationship stating 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 this content*". We usually express the amount of domain content through the *domain architecture*. We assume that we receive the domain content (or *domain artifacts*) through DA. By the domain content, we conceive the physical domain objects (entities), their features and relationships (between objects as well as features). For example, for a concrete domain, such as hardware design, the domain content consists of the following items: transistors, gates, flip-flops, etc., and their connections (signals), which are combined together accordingly to appropriate rules into a higher-level physical system with specific characteristics.

Our standpoint is that a relationship between the domain architecture and technological factors can be presented in the four-dimensional space as a 4D model (Fig. 1). We assume that the domain content is independent from the introduced abstractions. This model should be interpreted as follows. Each constituent part of the model (structuring,

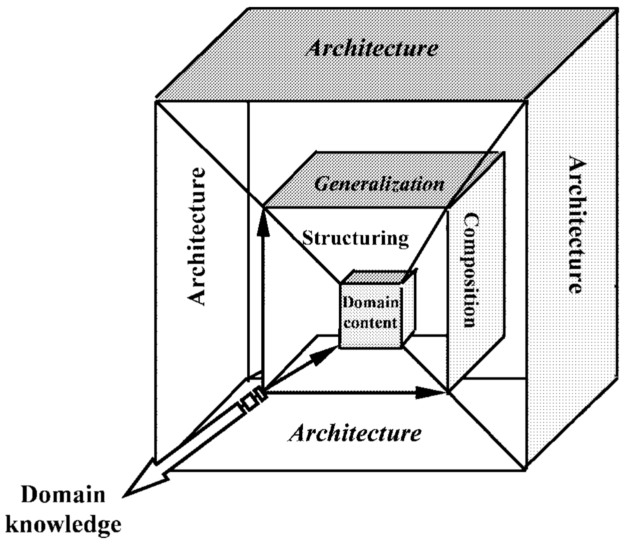

Fig. 1. The 4D relationship model between domain content, technological factors and architecture.

generalization, composition, and architecture) can be analyzed with respect to the different aspects. According to our model, the *domain knowledge* must be *projected* or *mapped* from *domain content* onto the *domain architecture*. To accomplish such mapping procedure, we need to pass through three abstraction layers (technological factors, which are represented as a middle cube). We can conceive that the axes of the *hypercube* (see Fig. 1) represent this mapping procedure. We represent architecture by the faces of the external cube. This can be interpreted as the fact that architecture itself may have various views. This standpoint is approximately the same as presented by D'Souza and Wills (D'Souza *et al.,* 1999, see Table 12.1, 484 p.).

Software technology can be characterized by a number of *technological factors*, which play a fundamental role in developing software. These factors reflect a particular aspect of software developing process. The well-known examples include reuse, optimization, integration, encapsulation, separation of concerns, adaptation, parameterization, etc. We argue that all technological factors can be divided into three main categories: *structuring, composition,* and *generalization* (see Fig. 2).

Structuring in this paper is understood as a process of dividing a particular integral concept into a hierarchy of interrelated structures (abstractions), which express important properties of that concept from different points of view. Structuring, in particular, deals with *structural* issues in software engineering, such as abstraction, integration, encapsulation, etc. It plays two roles: (1) *conceptually* – it represents a complex internal structure of a domain content as a hierarchy of interrelated abstractions; and (2) *operationally* – it paves the way for the creation of tools that work with these abstractions. As a result of structuring an abstract idea of domain content takes a concrete shape of component models, programming languages and programming paradigms, which are used for expressing this domain content. These structures are not accidental, but reflect the

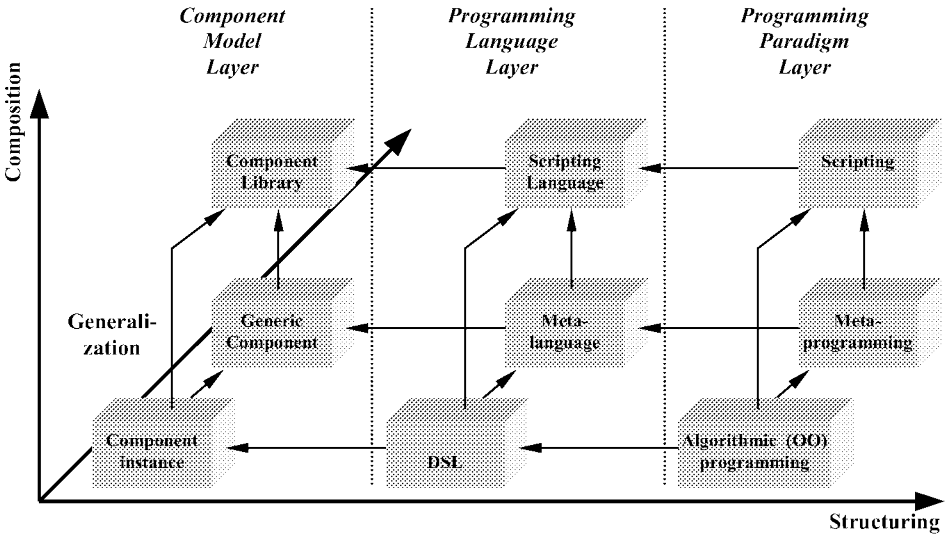

Fig. 2. Relationship model between structuring, generalization and composition.

up-to-date achievements in software engineering (*multi-design paradigm*).

At the lowest hierarchical layer of *component models*, a notion of component instance is introduced. A *component instance* is a direct representation of a domain entity in a symbolic (i.e., linguistic) form. In other words, a component is a consciously created symbol, which concretizes an abstract entity it represents. This *symbolization* of a domain content (1) represents a domain reality, though in a depleted way, in a concrete human-readable form; and (2) allows defining actions to be performed over it (i.e., the context of usage). Each component has a particular internal structure and implements a specific behavior.

In order to be understandable, all components, which represent entities of the same domain, are usually described at a higher layer of programming languages. The language, used for describing functionality of domain entities, is usually a DSL (e.g., VHDL for hardware design domain). Programming languages have two roles: (1) they introduce a standard set of symbols (the syntax of the language) and actions associated with these symbols (the semantics of the language) that unambiguously express a domain content; and (2) they allow creating tools (e.g, compilers, processors) that interpret these sets of symbols.

The process of programming, i.e., using a programming language to develop components which represent domain entities, is a sophisticated and error-prone process. In order to simplify and automate it, the methods of programming (i.e., algorithms, design patterns) are introduced at a higher layer of *programming paradigms*. These methods (1) extract and encapsulate common methods of solution of particular problems; and (2) allow creating tools (e.g., generators) that automate the process of programming.

We understand c*omposition* as a process of constructing a particular item from a collection of the lower-level items. There are two types of composition: (1) *physical composition*, where the code of composed components is actually *glued* together; and (2) *logical composition*, where the composed components *communicate* between themselves through some kind of ports, communication channels or connectors. The level of composition can be expressed by a number of lower-level items a higher-level item is composed of. The composition allows (1) to represent the system of hierarchical relations that link domain entities; and (2) to reuse the existing component instances in a target system. The role of composition is clearly seen at the component model layer, where a *component library* can be composed of generic components or component instances. Accordingly, *scripting languages* describe the syntax and semantics of composition, and *scripting paradigm* describes the rules and methods for implementing composition.

The *generalization* is a process of unifying similar items into a single item, which encapsulates their similarities and differences. The process differs from composition, because no new functionality is created, instead any of the involved parts can be derived at any time. Generalization is closely related to parameterization. In fact, parameterization is a part of generalization process, and constitutes of finding variable *aspects* of "look-alike" components, which are represented as generic parameters in the generic components' interface. The level of generalization can be naturally expressed by a number of generic parameters. Generalization allows (1) to encapsulate related domain concepts thus simplifying, both quantitatively and qualitatively, the design space; and (2) to reduce the size and improve the structure of the component libraries. Generalization, according to Sametinger (1997), can be performed using 4 techniques: (1) *widening*, (2) *narrowing*, (3) *isolating*, and (4) *configuring* component functionality. The result of generalization at a component model layer is a *generic component*, which represents in a compact way a set of "look-alike" component instances. Accordingly, *meta-languages* describe the syntax and semantics of generalization, and *meta-programming* paradigm defines the rules and methods for implementing generalization.

Programming languages are mapped onto three component models: the *component instance*, *generic component*, and *component library* ones. An instance implements the functionality of a single domain entity using an *algorithmic language*. A generic component encapsulates the similar functionality of the family of domain entities using *algorithmic & meta-languages*. A library encompasses a collection of generic components covering the entire domain using *algorithmic & meta- & scripting* languages.

Finally, in Fig. 3 we present the "*content-knowledge-architecture*" relationship as a part of the 4D model. We assume that the domain knowledge should be captured through DA.

All parts of the model express the *domain knowledge*, in one way or another. The component model captures the *semantics* of a domain. The *syntax* of a domain is reflected in the programming language. The *patterns* and *algorithms* of the domain phenomena and relationship between domain entities are established at the programming paradigm level. The *concepts* of a programming paradigm are implemented by a programming language, which describes the *behavior* of a component. The design solutions at the structural level are tightly coupled with overall domain architecture by the *rules of organization*.

Next, we evaluate the proposed model with respect to the known generative approaches.

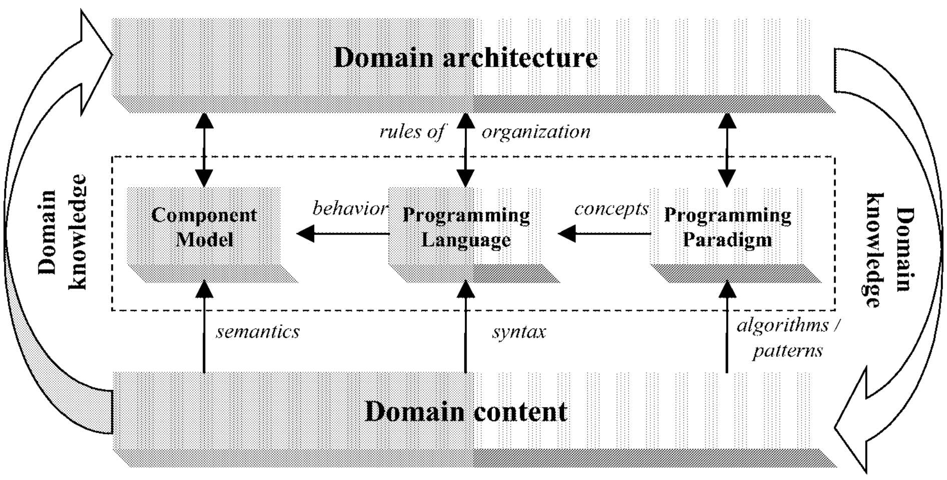

Fig. 3. Other view to the "content-knowledge-architecture" relationship model.

6. Evaluation of the Generator Models

Though the analyzed generative approaches are employed in considerably different domains (Bassett's approach – for information systems design; Batory's – networking, database applications, etc.; ours – hardware design), they reveal a great deal of similarity, which is reflected in the proposed 4D model. All approaches (see Table 1) use a hierarchical structure (Bassett's *hierarchy of frames*; Batory's *layers of abstraction*; ours *H-architecture*) which naturally expresses the complexity of the underlying domain content. This hierarchy is build using at least two programming paradigms: the *algorithmic* (OO) one for describing domain functionality, and the *meta-programming* (or *scripting*) one for generalization and textual modification. These paradigms are implemented using (1) *two* programming languages (Bassett's *Frame Command Language* + TL (e.g., Java, COBOL); ours *Open PROMOL* + TL (e.g., VHDL)), the first of which is domain-specific and independent of the second one; or (2) an extension of a standard language (Batory's P++ is an extension of C++). In the latter case, C++ is used for implementing domain algorithms, and the extension is used for describing generic layers of abstraction. At the component model layer, the similarity is even greater. All three approaches use parameterized software modules (Bassett's *frame*; Batory's *realm*; ours *generic component*) for expressing the similarities in the domain. These modules can form libraries, which capture the domain knowledge.

Summarizing, we argue that the proposed 4D model encompasses the analyzed generative approaches and clarifies their main features. However, we do not claim that this model expresses all features of software generators.

Table 1

Comparison of generative approaches

Approach	**Abstraction**			
	Component Model	**Programming Language**	**Programming Paradigm**	**Generator Architecture**
Bassett's	**Frame** – a parameterized software module	**Frame Command Language** – a meta-language for a textual manipulation	**Meta-programming** – frame commands perform a pure lexical manipulation of the target program text	**Hierarchy of Frames**
Batory's	**Realm** – an abstract parameterizable domain entity	**P++** - an extension of C++ for describing generic layers of abstraction	**Object-Oriented / Meta-programming** – layers are implemented as encapsulated suites of interrelated classes	**Layers of Abstraction**
Ours	**Generic Component** – a parameterized domain entity	**Open PROMOL** – a scripting language for target program **VHDL**, **C++**, etc.) modification	**Meta-programming / scripting** – GCs are imple-mented as textual composition of TL, SL code & other GCs	**M-, H- & G-architectures**

7. Discussion and Conclusions

While not a silver bullet, an abstraction plays a key role for creating components and generators. The programming languages, and domain-specific languages in particular, are the basic abstractions for expressing the domain content. The more complex domain content is, the higher-level of abstraction we need to describe the content. The high level standard languages, like C++ and VHDL, have higher-level constructs with the *meta-programming* capabilities (e.g., templates, generics) allowing to implement generic components.

The flexibility of generalization can be enhanced significantly either through extensions of the standard target languages or through the usage of the external scripting languages (*multi-language paradigm*). The generic components are designed using the different kind of abstractions, and thus allow expressing the families of domain instances concisely. Using various mechanisms of instantiation and generation, generic components can be combined together to form a system with the specific requirements. The component-based generators allow producing complex systems, and predominate over other generative technologies now.

The proposed 4D model of a domain generator is the result of the multi-paradigm design analysis over the domain of software generators. We argue that hardly any component-based domain generator can be built without considering three programming paradigms: (1) the *algorithmic* (OO); (2) *meta-programming*; and (3) *scripting* paradigms, which are implemented through the corresponding languages. These are: the *algorithmic*, *meta-programming*, and *scripting languages.* The first one addresses domain functionality. The second one deals with the generalization of similar ("look-alike") functionality. The third one deals with the composition of components as well as target systems. We argue that this model is manifested in the known software generator models, too.

Acknowledgements. Authors thank to the anonymous reviewer whose comments served for improving the paper.

References

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

Batory, D. (1997). Intelligent components and software generators, invited presentation to the software quality institute symposium on software reliability, Austin, Texas, April 1997. *Technical Report 97-06*, Department of Computer Sciences, University of Texas at Austin.

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

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

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

Budinsky, F. J., M.A. Finnie, J.M. Vlissides, P.S. Yu (1996). Automatic code generation from design patterns. *IBM Systems Journal*, **35**(2), 151–171.

Coplien, J.O. (2000). *Multi-Paradigm Design*. Ph.D. Thesis. Vrije Universiteit, Brussels, Belgium.

Coste, P., F. Hessel, Ph. Le Marrec, Z. Sugar, M. Romdhani, R. Suescun, N. Zergainoh, A.A. Jerraya (1999). Multilanguage design of heterogeneous systems. *CODES'99*, Rome, Italy, 54–58.

Czarnecki, K., U. Eisenecker (2000). *Generative Programming: Methods, Tools and Applications*. Addison–Wesley.

Czarnecki, K., U. Eisenecker, R. Gluck, D. Vandevoorde, T.L. Veldhuizen (1998). Generative programming and active libraries. In *Proceedings of the 1998 Dagstuhl-Seminar on Generic Programming*.

Deursen van, A., P. Klint, J. Visser (2000). Domain-specific languages: an annotated bibliography. *SIGPLAN Notices*, **35**(6), 25–36.

D'Souza, D.F., A.C. Wills (1999). *Objects, Components, and Frameworks with UML*. Addison–Wesley.

Edwards, S. H. (1999). The state of reuse: perception of the reuse community. *Software Engineering Notes*, **24**(3), 32-36.

Eisenecker, U.W. (1997). Generative programming (GP) with C++. In *Proceedings of the Joint Modular Language Conference' 97, Lecture Notes in Computer Science*, **1204**, Springer.

Gamma, E., R. Helm, R. Johnson, J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison–Wesley.

Horspool, R., M. Levy (1993). Translator-based multiparadigm programming. *Journal of Systems and Software*, **25**(1), 39–49.

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.

Jerraya, A.A, R. Ernst (1999). Multi-language system design. In *DATE'99*, Münich, Germany, March, pp. 696–701.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J. Irwin (1997). Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming. Lecture Notes in Computer Science*, **1241**, pp. 220–242.

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

Meiyappan, S., K. Jaramillo, P. Chambers (1999). 10 tips for generating reusable VHDL. *EDN*, August 19, 49–62.

Musser, D.R., A. Stepanov (1994). Algorithm-oriented generic libraries. *Software Practice and Experience*, **24**(7), 623–642.

Nierstrasz, O., T.D. Meijler (1995). Requirements for a composition language. In Ciancarini, P., O. Nierstrasz, A. Yonezawa (Eds.), *Object-Based Models and Langages for Concurrent Systems*, pp. 147–161.

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

Paige, R. (1997). Future directions in program transformations. *ACM SIGPLAN Notices*, **32**(1), 94–98.

Perry, D.E., A.L. Wolf. (1992). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, **17**(4), 40–52.

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

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

Spinellis, D.D. (1994). *Programming Paradigms as Object Classes: a Structuring Mechanism for Multiparadigm Programming*. Ph.D. Thesis. University of London, U.K.

Stohmann, J., E. Barke (1996). An universal CLA adder generator for SRAM based FPGAs. *FPL,* 44–54.

Stohmann, J., E. Barke (1997). A univeral pezaris array multiplier generator for SRAM based FPGAs. *ICCD,* 489–495.

Szyperski, C. (1999). *Component Software Beyond Object-Oriented Programming*. Addison–Wesley.

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

Štuikys, V., R. Damaševičius, E. Toldinas, G. Ziberkas, G. Majauskas, A. Venčkauskas (2000b). Applicability of open PROMOL for generic component design, modification and specialization. *Information Technology and Control*, **3**(16), 37–53.

Štuikys, V., G. Ziberkas, R. Damaševičius, G. Majauskas (2001a). Two approaches for developing generic components in VHDL. *Microelectronics Journal*, **2049**, Elsevier Science Ltd., Oxford.

Štuikys, V., R. Damaševičius, G. Ziberkas (2001b). Open PROMOL: an experimental language for target program modification. In *Forum on Design Languages (FDL'2001)*, Lyon, France.

Thibault, S., C. Consel (1997). A framework for application generator design. In *Proceedings of the 1997 Symposium on Software Reusability (SSR '97)*, ACM, pp. 131–135.

Veldhuizen, T.L. (1995). Using C++ template meta-programs. *C++ Report*, **7**(4), 36–43.

Villarreal, E.E., D. Batory (1997). Rosetta: a generator of data language compilers. *ACM*, 146–156.
Zimmermann, R. (1998). VHDL library of arithmetic units. In *Proceedings First Forum on Design Languages (FDL'98)*, Lausanne, pp. 267–272.

V. Štuikys received Ph.D. degree from Kaunas Politechnic Institute in 1970. He is an Associate Professor at Software Egineering Department, Kaunas University of Technology, Lithuania. His research interests include domain-specific reuse, high level domain-specific languages, expert systems and CAD systems, including VLSIC design based on high-level hardware description languages and software generation and program transformation.

R. Damaševičius received a bachelor degree in informatics from Kaunas University of Technology in 1999 and MSc degree in 2001. Currently he is Ph.D. student at Informatics faculty, Kaunas University of Technology. His research interests include software reuse, scripting and programming languages, design with VHDL, and software generation and program transformation.

Abstrakcijų, naudojamų kuriant srities generatorius, sąryšio modelis

Vytautas ŠTUIKYS, Robertas DAMAŠEVIČIUS

Šiame straipsnyje mes analizuojame abstrakcijas, vartojamas kuriant komponentinius srities generatorius. Šios abstrakcijos yra: programavimo paradigmos, kalbos, komponentų modeliai ir generatorių architektūrų modeliai. Atliktos analizės pagrindu mes pateikiame bendrą sąryšio tarp srities turinio, technologinių faktorių (struktūrizavimo, komponavimo ir apibendrinimo) ir srities architektūros modelį. Mes teigiame, jog šis modelis atspindi žinomus programų generatorių modelius.