ISSN 1392 – 124X INFORMATION TECHNOLOGY AND CONTROL, 2008, Vol. 37, No. 2

TAXONOMY OF THE FUNDAMENTAL CONCEPTS OF METAPROGRAMMING

Robertas Damaševičius, Vytautas Štuikys

Software Engineering Department, Kaunas University of Technology
Studentų St. 50, LT-51368 Kaunas, Lithuania

Abstract. Although widely used in software engineering, metaprogramming is often misunderstood. The researchers often disagree what concepts characterize metaprogramming. The concepts of metaprogramming are often used without acknowledging the usage of metaprogramming itself. We overview the examples and definitions of metaprogramming in computer science, identify, describe and discuss the fundamental concepts of metaprogramming (code generation, transformation, reflection, generalization, metaprogram, metadata, level of abstraction and separation of concerns). We analyze their relationship and present taxonomy, based on a study of sources on metaprogramming.

Keywords: metaprogramming concepts, taxonomy.

1. Introduction

Metaprogramming deals with the methods and processes of writing higher-level programs (metaprograms), which create other programs. Though *metaprogramming* [3, 11, 12, 19, 29, 48, 52, 53] is known for long and has been widely used in several areas of computer science [2, 27, 28, 37, 39], the term itself is often misunderstood. In many cases, the term 'metaprogramming' is used to denote different though implicitly related software engineering concepts. The concepts of metaprogramming are often used without acknowledging the usage of metaprogramming [4, 6, 7, 8, 33, 49, 50, 55]. This miscommunication between researchers and practitioners and misunderstanding of metaprogramming can be related to the fact the concepts of metaprogramming has not been extensively analyzed and categorized so far. Furthermore, metaprogramming is a much more loose approach than, e.g., object-oriented programming, which also exacerbates the difficulties of understanding, adoption and systematic application of metaprogramming.

Metaprogramming is widely used in the software development cycle, where it plays an essential role in program processors, interpreters, or compilers. Metaprogramming as a conceptual approach continues to evolve, and its principles are adapted to the ever-higher levels of abstraction. Examples include: *metamodeling* [1], *metadesign* [20, 16], *model-driven engineering (MDE)* [38] and *metaengineering* [34]. As the pursuit for increased productivity in software engineering continues, the role of metaprogramming is only destined to increase. However, metaprogramming is almost never consciously and explicitly integrated in the software development processes. Gaining awareness of its role is required to achieve progress in this domain. Understanding what concepts characterize metaprogramming is important to both practitioners, who are aiming to adopt novel design methods, as well as researchers.

2. Metaprogramming in computer science

2.1. Metaprogramming applications

Metaprogramming was known and used for a long time, especially in *formal logic programming* [39]. Now the scope of the application of the metaprogramming techniques is much wider such as programming language implementation, including compiler generation [49], application and software generators [5], product lines [6], generic component design [7], program transformations [29], program evaluation and specialization [23], generative reuse [8], software maintenance, evolution and configuration [13], middleware applications [12], XML-based web applications [28]. Applications of metaprogramming include compiler construction, BNF (*Backus-Naur Form*) used in compiler generators such as Lex and Yacc [26], macros, code analyzers (parsers), higher-order functions in logic metaprogramming [39, 52, 53], recursion, reflection [2, 30] including introspection and intercession [14], meta-classes [24], meta-object protocols [9], template metaprogramming [51], anticipatory optimization [8], mixin-based programming [40], design patterns [19], scripting [33], partial evaluation [23], web component deployment [28], and markup languages.

Many, if not all of the presented cases, can be summarized as *multi-stage programming* [47], i.e., developing programs in several different stages. Other approaches such as *parameterized programming* [22], *generative programming* [14], *generic programming* [17, 31], *reflection-oriented programming* [41] are very similar or use the same concepts as metaprogramming. Furthermore, metaprogramming techniques closely relate to novel software development technologies such as *aspect-oriented programming* [25].

2.2. Definitions of metaprogramming

There are many viewpoints, metaprogramming can be analyzed from, such as abstraction, languages, tools, programming techniques. Perhaps, the most popular is the abstraction-based view explained below.

Software systems typically consist of several levels of abstraction such as machine (or object) code, assembly code, algorithmic or object-oriented language code, pre-processor directives, etc. From the perspective of the abstraction level, metaprogramming means programming at a higher level of abstraction. Cordy and Shukla [11], for example, give the following definition. Metaprogramming is '*the technique of specifying generic software source templates from which classes of software components, or parts thereof, can be automatically instantiated to produce new software components*'. A metalanguage, which is a mechanism for introducing a higher-level of abstraction, does not appear in this definition. It is assumed that source templates (such as C++ templates) are higher-level generic abstractions of the source (or domain) language itself.

Another definition introduces a concept of a metalanguage explicitly: '*any language or symbolic system used to discuss, describe, or analyze another language or symbolic system is a metalanguage*' [14]. A program written in a metalanguage is a metaprogram. According to [6], a metaprogram is '*a program that generates the source of the application ... by composing pre-written code fragments*'. Examples of metaprograms are application generators, and building application generators such as parser generators. Metaprogramming then can be defined as '*creating application programs by writing programs that produce programs*' [27].

Sheard [39] emphasizes the role of metaprogramming in program generation explicitly. He says that in a metaprogramming system, '*metaprograms manipulate object-programs*'. A metaprogram is a program, which '*may construct object-programs, combine object-program fragments into larger object-programs, observe the structure and other properties of object-programs*'. A similar definition is given by F. Rideau: '*Metaprogramming, the art of programming programs that read, transform, or write other programs*' [37], and J. Bartlett: '*Metaprogramming is writing programs that themselves write code*' [3].

2.3. The usage of metaprogramming

A common usage of metaprogramming is to provide mechanisms for writing generic code, i.e. explicitly implementing generalization in the domain. Domain language implements commonalities in a domain, while a metalanguage enables developers to specify variations to be implemented in the domain system, and to synthesize customized implementations by composing the domain code fragments.

The genericity is usually achieved by the *parameterization of differences* in different program representations, which enables representing components with many commonalities in a compact way. This simple feature of metaprogramming enables reusability to be substantially improved by providing parameterized components, which can be instantiated into target programs for different choices of parameters.

The basis of metaprogramming is a separation of the domain artefacts from the knowledge of how to customize and glue them together. The higher-level program (metaprogram) uses pieces of lower level constructs as data. This enables generalization and automatic creation of the customized programs.

Metaprogramming can be implemented in several ways. At the abstraction level, we need to analyze the capabilities of the language and separate the concerns, which relate to implementing the basic functionality, from those which allow expressing generic solutions and customized specifications. This separation may be accomplished, for example, implicitly using only the internal capabilities of a given domain language, or explicitly either introducing some extensions to the domain language or using an external metalanguage.

The product of metaprogramming is a *metaprogram* (or *metaspecification*), which describes a family of the related (generic) functionality in a narrow well-defined domain. Thus, a metaprogram together with its environment is a domain program generator. Formally, the goal of metaprogramming is to create a metaprogram for a given domain of application. Summarizing, metaprogramming is a higher-order programming technique that is used for achieving generalization via manipulation with other program structures.

3. Analysis of metaprogramming sources

A wide variety of sources (books, journals, conference proceedings), selected from IEEE Xplore online database, published from 1965 to 2007 and related to metaprogramming (i.e., had such keywords as 'metaprogramming' or 'meta-level programming'), were reviewed from different viewpoints (computer science, information systems, software engineering).

The analysis consisted of reviewing each source document for the identification of specific concepts as metaprogramming concepts. The concepts from 41 sources were recorded. There were 35 concepts mentioned as belonging to the metaprogramming approach

(such as manipulation, code generation, etc.). Table 1 presents the numerical frequency of the concepts in the analyzed literature sources (note that our analysis is by no means exhaustive). Since many different yet equivalent terms are used, we had to group the concepts with the similar meaning into groups or classes. Of the 35 concepts, 8 concept classes were identified by the majority (73%) of the sources: transformation (including manipulation and other synonymous terms), code generation, reflection, generalization, metaprogram (including generic component, template, macro etc.), metadata, level of abstraction (including various aspects of representation) and separation of concerns. We analyze these concepts, which we consider as fundamental to metaprogramming, in Section 4.

Table 1. Concepts related with metaprogramming

Concept class	Concept	No.	Total no.	Percentage
Transformation	Manipulation	11	**29**	**70%**
	Transformation	7		
	Modification	5		
	Adaptation	4		
	Translation	1		
	Preprocessing	1		
Generation	Code generation	14	**17**	**41%**
	Instantiation	2		
	Weaving	1		
Metaprogram	Template	6	**16**	**39%**
	Generic component	5		
	Macro	2		
	Metaprogram	2		
	Metaspecification	1		
Levels of abstraction	Representation	6	**11**	**27%**
	Abstraction	4		
	Encapsulation	1		
Generalization	Construction	5	**8**	**20%**
	Generalization	2		
	Parameterization	1		
Separation of concerns	Analysis	5	**8**	**20%**
	Concern separation	3		
Reflection	Reflection	6	**7**	**17%**
	Introspection	1		
Metadata	Metadata	3	**5**	**12%**
	Parameters	2		
Other concepts	Metaobject protocol	2	**11**	**27%**
	Traits	2		
	Theorem proving	1		
	Partial evaluation	1		
	Inspection	1		
	Specialization	1		
	Runtime execution	1		
	Optimization	1		
	Interpretation	1		

4. Fundamental concepts of metaprogramming

4.1. Transformation

Program transformation can be defined as '*the derivation of programs from formal specifications, and the derivation of new program versions from old program versions*' [36]. Program transformation can be described using a higher-level language such as BNF used in compiler generators [49], or Open PROMOL [44], or any other metalanguage. It is used for the derivation of programs from high-level specifications or older program versions in a semantics preserving way.

Generally, program transformation is a manipulation of its representation resulting in the change of the form (syntax) of the program. Its semantics may be changed or not in the process. A step-wise manipulation, which (1) is defined on a programming language domain, (2) uses a formal model to support the refinement, and (3) simultaneously preserves the semantics, is known as a *formal program transformation* [36]. The general case of program transformation, however, does not require the definition of a formal model, as well as has no restrictions on the changes of semantics.

Conventional programming is oriented at developing tools for manipulating data, i.e. data processing, representation, visualization, communication, etc. The inputs to a program are data structures. The output is the resulting data produced by the program in the variety of forms dependant upon a given application. The produced result is used then as-is by other programs or the user. Metaprogramming, on the other hand, is oriented at developing tools for manipulating with lower-level programs, i.e., automatic analysis (parsing), automatic adaptation (modification) of a program to the context of usage, generation of instances, etc., which all can be summarized as *transformation*.

Our definition of program transformation is as follows. Program transformation is the process of changing one form of a program (source code, specification or model) into another, as well as a formal or abstract description of an algorithm that implements this transformation [46]. The role of transformation in metaprogramming is that the transformation algorithm describes generation of a particular instance depending upon values of the generic parameters. The transformation algorithm ranges from simple metaconstructs such as *meta-if* (conditional generation) and *meta-for* (repetitive generation) to the sophisticated application-specific metapatterns, which are composed of the nested combinations of the simpler metaconstructs.

4.2. Generation

Software generation is an automated process of creation of a target system from a high-level

specification [50], such as a metaprogram. Code generation is the process by which a code generator converts a syntactically-correct high-level program into a series of lower-level instructions. The input to the code generator stage typically consists of a parse tree, abstract syntax tree, or intermediate language code. Since the target machine may be a physical machine such as a microprocessor, or an abstract machine such as a virtual machine or an intermediate language, the output of code generator could be in any language.

In a more general sense, code generation is used to produce programs in some automatic manner, thus reducing the need for programmers to write code manually. Code generation can be done either at run-time, including load-time (e.g., just-in-time compilers that produce native code from byte-code), or compile-time (e.g., a compiler-compilers such as *yacc* [49]). A pre-processor is an example of the simplest code generator, which produces target code from the source code by replacing predefined keywords.

The role of code generation in metaprogramming is centred on the development of program generators, i.e. higher-level programs that generate other programs adapted for specific applications. The metalanguage processor manipulates with program instances or some parts of instances as well as with data structures. In general, the output is a family of the related program instances, or only one instance from the family.

4.3. Metaprogram

Metaprograms are described using generic constructs of high-level languages (such as templates in C++) or a different language (macro language, metalanguage). Other terms synonymous to ‘metaprogram’ such as metacomponent, metaspecification, etc. are also used. Metaprograms usually are generic and have a number of parameters; hence the name ‘generic component’ also is used (note that in this paper we do not distinguish between generic and generative component). A generic component can be defined as a software module allowing choosing its properties to a certain degree without necessarily having to write or change code manually [7].

Conceptually, a generic component abstractly and concisely represents a set of closely related (“look-alike”) software components with slightly different properties. Since it is sensible to integrate such components that share a considerable amount of common code in a generic component, generic components can be considered as a component family [39].

Generic components are not specific code fragments or common domain programs. Each generic component contains formal parameters and structures that allow it to be systematically modified to become any of a set of specific components (instances) [4]. Generic parameters together with their respective range of supported values are usually identified at the generic component's interface. Instantiating generic component means to choose actual values for the supplied generic parameters, and let the appropriate generator(s) to perform the necessary modifications.

Metaprogram is a generic component implemented using a metalanguage. Metaprogram represents a family of similar component instances and contains different functionality (variations) that can be instanttiated through parameterization in order to create a specific component instance. The role of a metaprogram in metaprogramming is the same as, e.g., of a class in the OO programming, i.e., it is a basic unit of abstraction for composing larger metaprogramming systems.

4.4. Levels of abstraction

Abstraction is the fundamental way of organizing knowledge and grouping similar facts together [43]. Abstraction hides unimportant details of implementation and emphasizes the information that is important for a developer or end-user. There are multiple levels (or layers) of abstraction in software, where each level represents a different model of the same information and processes, but uses a different semantic system of expression (or grammar) to express the content of a particular domain. Each higher (relatively abstract) level is built on a lower (relatively concrete) level.

What is usually common to all cases of metaprogramming is that there are two (or more) levels of abstraction. Each level of abstraction uses a different semantic system. The lower level of abstraction is usually domain-oriented and is used to describe common domain functionality using a domain language. The higher level of abstraction (generic or meta) is used for expressing variability in a domain and describing manipulations with the syntactic units of the lower level of abstraction using a specific metalanguage.

The levels of abstraction are semantic systems that are grouped together to represent different aspects of design in metaprogramming systems [45]. The role of levels of abstraction in metaprogramming is centred at the construction of a metaprogram. Metaprograms embody different (usually orthogonal) aspects of domain systems. Such aspects are implemented and composed by structuring domain programs in terms of modules or layers, which use different semantic systems and enable various functionalities to be added.

4.5. Generalization

Generalization provides a form of knowledge representation. A higher, more generalized (meta-level) of domain knowledge encapsulates an understanding of the general properties and behaviour possessed by a subset of its domain entities. Introduction of generalization means a transition to the higher level of abstraction, where domain knowledge can be represented and explained more comprehensibly and effectively.

In computer science, generalization is usually understood as a technique of widening of an object (component, system) in order to encompass a larger domain of objects (systems, applications) of the same or different type [16]. Generalization identifies commonalities and variability (variations) among a set of domain entities. The commonality may refer to essential features of a domain entity such as attributes or behaviour, or may concern only the similarity in syntactic description, while variability refers to the specific features pertaining to a specific domain component or program.

Therefore, generalization can be understood as a transformation of a specific domain component into a generic component (metaprogram) that is more widely usable and reusable than the original one. The role of generalization in metaprogramming is the development of a metaprogram using some specific domain program (component) as a basis, and involves capturing of the domain commonalities, while expressing the domain variations at a higher level of abstraction.

4.6. Separation of concerns

The term 'separation of concerns' was first introduced by E.W. Dijkstra in 1974 [18]. Separation of concerns at the conceptual level is generally considered as a primary means to manage domain complexity. The program parts related to the separated concerns are implemented separately, and integrated back to form a complete design. The principle of separation of concerns can be applied in various ways and is actually one of the key principles in software engineering [21]. This principle states that a given problem involves different kinds of concerns, which should be identified and separated to cope with complexity, and to achieve the required engineering quality factors such as flexibility, dependability, maintainability, and reusability.

Separation of concerns is the process of breaking a design problem into distinct tasks that are orthogonal and are implemented separately. Metaprogramming widely exploits the principle of separation of concerns, which is used to separate variable parts of the domain program from the fixed (common) parts. A metalanguage also should allow to separate clearly the computational (algorithmic, behavioural) aspects (i.e., the ones dealing with domain functionality) and compositional aspects (i.e., the ones dealing with component integration, interoperability, etc.), thus achieving a great deal of flexibility and reusability.

4.7. Reflection

Reflection is the ability of a program to manipulate with the state of the program (e.g., its semantics) as data during its own execution, or the ability to describe inside a language the semantics of generated programs [2]. Reflection is the ability of a program to observe and possibly modify its structure and behaviour [30]. Usually reflection refers to run-time or dynamic reflection, though some programming languages support compile-time (static) reflection. E.g., during compilation of source code, information about the structure of a program is usually lost. If a system supports reflection, the structure of a program may be preserved as metadata embedded with the compiled code. In this context, metaprogramming is a reflective activity, because it allows developing programs that can create other programs.

4.8. Metadata

Metadata are structured, encoded data that describe characteristics of information-bearing entities to aid in the identification, discovery, assessment, and management of the described entities [10]. Since their introduction in the 1970s, metadata have been the object of systematic research in such areas as data warehouse managing and the Web. Metadata can range from finite-state-machine models of a component to plain documentation. In fact, any software engineering artefact can be a metadatum for a given component, as long as (1) the component developer is involved in its production, (2) it is packaged with the component in a standard way, and (3) it is processed by automated development tools [32].

Often, metadata are shortly defined as a description of data [42]. In the context of metaprogramming, metadata are the descriptions of the properties or concerns of a specific layer of abstraction in a metaprogramming system. The role of metadata in metaprogramming is to describe and represent additional information about the meta level of abstraction in a metaprogram. Examples of metadata include descriptions of generic parameters in generic components, or the description of domain language syntax in compiler generators, or the structure of generated documents.

5. Taxonomy of the metaprogramming concepts

In addition to a lack of consensus on the fundamental concepts, the software engineers lack an understanding of how metaprogramming concepts can be classified to characterize the metaprogramming approach. There are few works on metaprogramming taxonomy. Several authors do summarize the basic concepts of metaprogramming, however, this is usually limited to one sentence. For example, '*metaprogramming involves analyzing, generating, and transforming object programs*' [55]. Only two taxonomies given in the literature are comprehensive (see Table 2).

The metaprogramming taxonomies presented by Sheard [39] and Pasalic [35] overlap considerably, though there are some differences. The most serious objection against these taxonomies is that these are not as much the taxonomies of metaprogramming concepts, as taxonomies of metaprogramming systems (generators) and tools (metalanguages). There are

multiple issues addressed in these taxonomies, such as the number of languages used in a metaprogramming system (*homogenous metaprogramming* – a metalanguage and a domain language are the same languages; *heterogeneous metaprogramming* – the ones are actually different languages), time of usage during software development cycle (*static* – before compilation / execution, *run-time* – during execution), dependence of a metalanguage upon domain language (*closed* – dependant, *open* – independent), separation of static and dynamic parts of a metaprogram (*manual, automatic*). Therefore, the classification of concepts in these taxonomies is opaque, many of important concepts such as abstraction or reflection are left out, while other issues that are not directly related to metaprogramming (such as open source code) are overemphasized.

Table 2. Known taxonomies of metaprogramming

Sheard's taxonomy [39]		**Pasalic's taxonomy [35]**	
Kind of metaprogram	Program generator	Kind of metaprogram	Program generator
	Program analyzer		Program analyzer
Separation of languages	Homogeneous	Separation of languages	Homogeneous
	Heterogeneous		Heterogeneous
Use time	Static	Type of metalanguage	Open
	Run-time		Closed
Separation of static and dynamic code of metaprogram	Manual		
	Automatic		

A new taxonomy of the metaprogramming concepts is proposed in Table 3. This taxonomy is more consistent and wider than known taxonomies (see Table 2). Some parts of known taxonomies can be also found in our taxonomy, e.g., types of metaprograms, use time, separation of concerns, usage of metadata. The novelty of the proposed taxonomy is that a hierarchy of concepts is introduced. All concepts are categorized either as structural concepts or process concepts. The identification of a relationship between structural and process concepts of metaprogramming is a complex task as it is summarized in Figure 1.

The structural concepts describe the basic abstractions (metaprogram, metadata) and principles of construction (separation of concerns, levels of abstraction) used while developing metaprogramming artefacts. Their properties are: 1) *static* (structure and capabilities are defined by the designer), 2) *construction-time* (used during construction of the metaprogramming systems and artefacts), and 3) *tool-dependant* (depend upon specific selection of a metalanguage, etc.). The process concepts describe basic operations and processes that are performed by a designer of the metaprogramming artefacts. They are: 1) *dynamic* (describe some method or process rather than a specific tool or abstraction), 2) *domain-independent* (can be implemented using different meta and domain abstractions and tools). Transformation, generation, reflection are used in compile-time or run-time, i.e., during processing or execution of a metaprogram, while generalization is used during creation of metaprogramming artefacts.

Table 3. Taxonomy of the fundamental concepts of metaprogramming

Concept class	**Concept**	**Equivalent terms used in the literature**	**Definition**
Structure	Metaprogram	Meta-component, meta-specification, metafunction, template, generic component, parameterized component	A generic component implemented using a metalanguage that represents a family of similar component instances and contains different functionality (variations)
	Levels of abstraction	Layers of abstraction	Semantic systems that are grouped together to represent different aspects of design in metaprogramming systems
	Separation of concerns	Separation of aspects, orthogonalization, 'divide-and-conquer'	The process of breaking a design problem into distinct tasks that are orthogonal and are implemented separately
	Metadata	Annotations	The description of the properties or concerns of a specific layer of abstraction in a metaprogramming system
Process	Transformation	Manipulation, modification, adaptation	The process of changing one form of a program into another
	Generation	Program/code generation	The creation of a target system from a high-level specification
	Reflection	Introspection, intercession	Ability of a program to observe and modify its structure and behavior
	Generalization	Parameterization	Transformation of a specific domain component into a generic component that is wider usable and reusable than the original one

Another interesting question is how structural concepts depend upon process concepts and vice versa. Starting from the construction of the metaprogramming artefacts, the designer analyzes the domain, gathers the available domain artefacts and requirements, and applies generalization. As a result, common and variable concerns of the domain are separated, the levels of abstraction are identified, metadata and metaprograms are created. These structural concepts are further used by other metaprogramming processes as follows. Transformation involves manipulation of domain language code using metaprograms, where manipulation algorithms are implemented at a higher level of abstraction and depend upon

metaparameters described as metadata. Generation means instantiation of a metaprogram and generation of domain code using specific values of the metaparameters. Reflection means the analysis of the domain level of abstraction, metadata extraction and passing to the metalevel of abstraction.

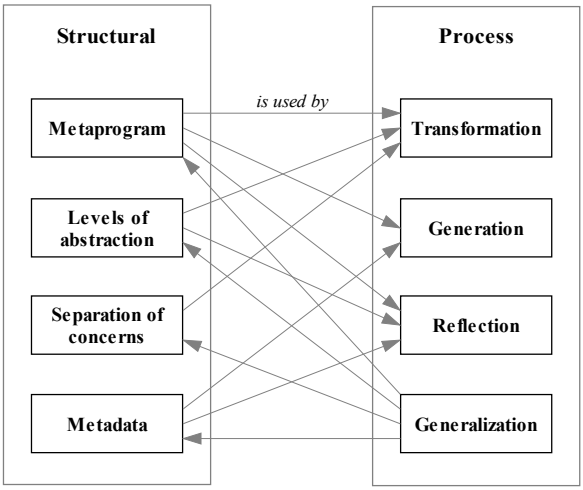

Figure 1. Relationship between structural and process concepts of metaprogramming

```
$
"Gate function" {AND, OR, XOR, NAND, NOR} f := AND;
"Number of gate inputs" {2..8} inp := 2;
"Width of data path"  {1, 8, 16, 32} width := 8;
$
ENTITY gate IS
  PORT(
    @gen[inp, {, }, {x}] :
        IN BIT@if[width>1, {_VECTOR(0 TO @sub[width-1])}];
    y : OUT BIT@if[width>1, {_VECTOR(0 TO @sub[width-1])}]
  );
END;

ARCHITECTURE behaviour OF gate IS
BEGIN
  y <= @gen[inp, { @sub[f] }, {x}];
END behaviour;                                          (a)
```

```
ENTITY gate IS
  PORT(
    x1, x2 : IN BIT_VECTOR(0 TO 7);
    y : OUT BIT_VECTOR(0 TO 7)
  );
END;

ARCHITECTURE behaviour OF gate IS
BEGIN
  y <= x1 AND x2;
END behaviour;                                          (b)
```

Figure 2. Example of the concepts of metaprogramming

See Figure 2 for an illustration of the structural concepts of metaprogramming using Open PROMOL [44] as a metalanguage and VHDL as a domain language. Figure 2a presents a metaprogram, which encapsulates a family of logic gates that have similar structure and functionality. The metaprogram has three different levels of abstraction: 1) *metainterface* (between '$' symbols), where metaparameters of the metaprogram are described, 2) *domain language layer*, which describes basic domain functionality that is common to all logic gates (described in VHDL), and 3) *metalanguage layer,* which describes the variability of the logic gate family (described in Open PROMOL).

Separation of concerns is illustrated in several ways: the interface of the metaprogram is separated from its implementation, the domain language layer is separated from the metalanguage layer, and each aspect of generalization is implemented using a separate metaparameter. Metadata concept is illustrated by the component's interface, where each metaparameter is annotated with its description.

The process class of metaprogramming concepts also can be seen in Figure 2a and 2b. Generalization is introduced via metaparameters in the metainterface of the metaprogram. Transformation (modification) is achieved via a set of Open PROMOL functions. The **@sub** function returns the value of a parameter. The **@if** function performs conditional generation. The **@gen** function generates look-alike strings. The result of code generation can be seen in Figure 2b, where an instance of logic gate family is shown. This instance is one of 140 different instances that can be generated from the metaprogram given in Figure 2a.

Reflection is a more difficult concept that requires parsing and analysis of source code. An example of reflection was demonstrated in [16], where parser automatically parses VHDL code and extracts component interface information that is further used to generate component wrappers for specific domain applications.

6. Evaluation and conclusion

Metaprogramming is a very powerful software engineering method that requires a systematic application to use properly, rather than to resort to its use at every opportunity. *Ad hoc* application of metaprogramming tends to make programs harder to understand, validate and maintain [54]. The major benefits of metaprogramming are software reuse and automated program development. A major stumbling block to achieving the benefits is the understanding and learning of the metaprogramming approach. One reason is that software designers do not thoroughly understand yet the fundamental concepts that define metaprogramming.

Programming requires that programmers understand fully the syntax, semantics, capabilities, and limitations of the languages that they program with. As metaprogramming usually means using two (or more) languages – domain language and metalanguage – in one specification or system at multiple levels of abstraction, the designer must learn at least twice as much of information. The metaprogrammer faces difficulties in understanding, programming in two languages simultaneously, and reading such *multi-language specifications* [15]. The metaprogrammer not only needs to know the details of how to program in domain-specific languages and metalanguages, but also the details of how they are implemented, how to communicate between them, and what sort of impeding mismatches there are between them.

The metaprogrammers and metadesigners should be domain experts that have extensive and thorough knowledge of domain content as well as metaprogramming methods and tools. Therefore, there is a need for thorough domain analysis, construction of domain vocabularies, taxonomies and development of domain knowledge ontologies for the metaprogramming domain. This study is a first step towards building ontology for the metaprogramming domain systematically. The study presents a taxonomy of the fundamental metaprogramming concepts that were identified from a sample of sources from the metaprogramming literature and organized into two groups: structural and process concepts. The identification of a relationship between these two sets of concepts is a complex task and covers the construction and usage of the metaprogramming artefacts. The results of this study should help software engineering researchers and practitioners to better understand, adopt and apply the methods of metaprogramming.

Future work will focus on the development of the comprehensive ontology of the metaprogramming domain. The construction of such ontology will allow providing a shared and common understanding of the metaprogramming domain, and will facilitate knowledge sharing between metaprogrammers.

References

[1] **C. Atkinson, T. Kuhne.** The role of meta-modeling in MDA. *In J. Bezivin, R. France (eds.)*, *Workshop in Software Model Engineering*, 2002.

[2] **G. Attardi, A. Cisternino.** Reflection support by means of template metaprogramming. *Proc. of Third Int. Conf. on Generative and Component-Based Software Engineering GCSE01*, LNCS, *Vol.*2186, *Springer-Verlag, Berlin*, 2001, 118-127.

[3] **J. Bartlett.** The art of metaprogramming. *IBM developerWorks, October* 2005, *http://www-128.ibm.com /developerworks/linux/library/l-metaprog1.html?ca= dgr-lnxw06MetaCoding.*

[4] **P. Basset.** Framing software reuse: lessons from the real world. *Yourdon Press, Prentice Hall*, 1997.

[5] **D. Batory, S. Dasari, B. Geraci, V. Singhal, M. Sirkin, J. Thomas.** Achieving reuse with software system generators. *IEEE Software, September* 1995, 89-94.

[6] **D. Batory.** Product-line architectures, Invited Presentation. *Smalltalk and Java in Industry and Practical Training, Erfurt, Germany*, 1998, 1-12.

[7] **M. Becker.** Generic components: a symbiosis of paradigms. 2*nd Int. Symp. on Generative and Component-Based Software Engineering GCSE* 2000, *Erfurt, Germany, October* 9-12, 2000, *LNCS Vol.*2177, *Springer*, 100-113.

[8] **T.J. Biggerstaff.** A Perspective of Generative Reuse. *Annals of Software Engineering* 5, 1998, 169-226.

[9] **S. Chiba.** A Metaobject Protocol for C++. *ACM SIGPLAN Notices* 30(10), 1995, 285-299.

[10] Committee on Cataloging Task Force on Metadata. *Summary Report* 1999, *http://www.libraries.psu.edu /tas/jca/ccda/tf-meta3.html.*

[11] **J.R. Cordy, M. Shukla.** Practical Metaprogramming. *Proc. of the 1992 IBM Centre for Advanced Studies Conference, Nov*. 1992, 215-224.

[12] **J.K. Cross, D.C. Schmidt.** Metaprogramming techniques for distributed real-time and embedded systems. *Proc. of 7th IEEE Int. Workshop on Object-Oriented Real-Time Dependable Systems, January* 7-9, 2002, *San Diego, CA, USA*, 3-10.

[13] **K. Czarnecki, U.W. Eisenecker.** Separating the configuration aspect to support architecture evolution. *Proc. of* 14*th European Conf. on Object-Oriented Programming* (ECOOP'2000), *Cannes, France, June* 11-12, 2000.

[14] **K. Czarnecki, U. Eisenecker.** Generative Programming: Methods, Tools and Applications. *Addison-Wesley*, 2001.

[15] **R. Damaševičius, V. Štuikys.** Separation of Concerns in Multi-language Specifications. *INFORMATICA, Vol*.13, *No*.3, 2002, 255-274.

[16] **R. Damaševičius.** On the Application of Meta-Design Techniques in Hardware Design Domain. *International Journal of Computer Science (IJCS)*, *Vol*.1, *No*.1, 2006, 67-77.

[17] **J.C. Dehnert, A.A. Stepanov.** Fundamentals of Generic Programming. *Report of the Dagstuhl Seminar on Generic Programming, Schloss Dagstuhl, Germany. LNCS Vol*.1766, 1-11.

[18] **E.W. Dijkstra.** Selected Writings on Computing: A Personal Perspective. *Springer-Verlag*, 1982.

[19] **D. von Dincklage.** Making Patterns Explicit with Metaprogramming. *Proc. of* 2*nd Int. Conf.e on Generative Programming and Component Engineering, GPCE* 2003, *Erfurt, Germany, September* 22-25, LNCS *Vol*. 2830, *Springer, Berlin/Heidelberg*, 2003, 287-306..

[20] **G. Fischer, E. Giaccardi, Y. Ye, A.G. Sutcliffe, N. Mehandjiev.** Meta-design: a manifesto for end-user development. *Commun. ACM* 47(9), 2004, 33-37.

[21] **C. Ghezzi, M. Jazayeri, D. Mandrioli.** Fundamentals of Software Engineering. *Prentice Hall*, 2003.

[22] **J.A. Goguen.** Parameterized programming and software architecture. *Proc. of* 4*th Int. Conf. on Software Reuse, ICSR-4, Orlando, USA,* 23–26 April 1996, 2–11.

[23] **N.D. Jones, C.K. Gomard, P. Sestoft.** Partial Evaluation and Automatic Program Generation. *Prentice Hall International, June* 1993.

[24] **G. Kiczales, J. des Rivieres, D.G. Bobrow.** The Art of the Metaobject Protocol. *MIT Press*, 1991.

[25] **G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J. Irwin.** Aspect-oriented programming. *Proc. of the European Conf. on Object-Oriented Programming (ECOOP'*1997*). LNCS* 1241, *Springer-Verlag*, 1997, 220-242.

[26] **J.R. Levine, T. Mason, D. Brown.** Lex and Yacc. *O'Reilly and Associates, Inc*., 1992.

[27] **L.S. Levy.** A metaprogramming method and its economic justification. *IEEE Transactions on Software Engineering,* 12(2), 1986, 272-277.

[28] **W. Löwe, M. Noga.** Metaprogramming applied to web component deployment. *Electronic Notes in Theoretical Computer Science,* 2002, 65(4).

[29] **A. Ludwig, D. Heuzerouth.** Metaprogramming in the large. *G. Butler and S. Jarzabek (Eds.), Generative and Component-Based Software Engineering. LNCS Vol*. 2177, *Springer*, 2001, 178-187.

[30] **J. Malenfant, M. Jaques, F.-N. Demers.** A tutorial on behavioral reflection and its implementation. *Proc. of the Reflection* 96 *Conference*, *April* 1996, *San Francisco, CA*, 1-20.

[31] **D.R. Musser, A.A. Stepanov.** Generic Programming. *Proc. of Int. Symp. on Symbolic and Algebraic Computation ISSAC'*88, *Rome, Italy, July* 4-8, 1988. *LNCS Vol.*358, *Springer* 1989, 13-25.

[32] **A. Orso, M. J. Harrold, D. S. Rosenblum.** Component metadata for software engineering tasks. *Proc. of* 2*nd Int. Workshop on Engineering Distributed Objects, EDO* 2000, *Davis, CA, USA, November* 2-3, 2000. LNCS *Vol*.1999, *Springer,* 129-144.

[33] **J.K. Ousterhout.** Scripting: Higher Level Programming for the Century. *IEEE Computer* 31(3), 1998, 23-30.

[34] **K.D. Palmer.** Vajra Logic and Mathematical Metamodels for Meta-Systems Engineering: Notes on the Foundations of Emergent Meta-Systems Theory and Practice. 12*th Annual Int. Symp. of the Int. Council On Systems Engineering (INCOSE)*, *Las Vegas, NV, USA*, 28 July - 1 *August* 2002.

[35] **E. Pasalic.** The Role of Type Equality in Meta-Programming. *PhD thesis, Oregon Health and Sciences University*, 2004.

[36] **A. Pettorosi.** Future Directions in Program Transformation. *ACM Computing Surveys, Vol*.28, No.4, *December* 1996, 171-174.

[37] **F. Rideau.** Metaprogramming and Free Availability of Sources. *Proc. of Autour du Libre Conference, Bretagne*, 1999.

[38] **D.C. Schmidt.** Model-Driven Engineering. *IEEE Computer* 39 (2), 2006, 25-31.

[39] **T. Sheard.** Accomplishments and research challenges in metaprogramming. 2*nd Int. Workshop on Semantics, Application, and Implementation of Program Generation (SAIG'*2001*),* F*lorence, Italy. LNCS Vol.* 2196, Springer, 2001, 2-44.

[40] **Y. Smaragdakis, D. Batory.** Mixin-Based Programming in C++. 2*nd Int. Symp. on Generative and Component-Based Software Engineering (GCSE'* 2000*), Erfurt, Germany, October* 9-12, 2000. LNCS *Vol*. 2177, Springer, 2000, 163-177.

[41] **J.M. Sobel, D.P. Friedman.** An Introduction to Reflection-Oriented Programming. *Proc. of Reflection* 96, *San Francisco, CA, USA, April* 1996, 107-126.

[42] **D. Soltes.** Metadata and Metainformation – Old Concepts and New Challenges. *IASSIST QUARTERLY Vol*. 23, 1999, 12-14.

[43] **A. Stepanov.** Future of Abstraction. *A keynote address at Joint ACM Java Grande – ISCOPE* 2002 *Conference, Seattle, Washington, November* 3-5, 2002.

[44] **V. Štuikys, R. Damaševičius.** Scripting Language Open PROMOL and its Processor. *INFORMATICA Vol*.11, *No*.1, 2000, 71-86.

[45] **V. Štuikys, R. Damaševičius.** Relationship Model of Abstractions Used for Developing Domain Generators. *INFORMATICA Vol*.13, *No*.1, 2002, 111-128.

[46] **V. Štuikys, R. Damaševičius.** Taxonomy of the Program Transformation Processes. *Information Technology & Control, No*. 1 (22), 2002, 39-52.

[47] **W. Taha.** Multi-Stage Programming: Its Theory and Applications. *PhD thesis, Oregon Graduate Institute of Science and Technology*, 1999.

[48] **J. Templ.** Metaprogramming in Oberon. *PhD Dissertation. ETH Zürich*, 1995.

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

[50] **S. Thibault, C. Consel.** A Framework for Application Generator Design. *ACM SIGSOFT Software Engineering Notes* 22(3), 1997, 131-135.

[51] **T.L. Veldhuizen.** Using C++ template metaprograms. *C++ Report* 7(4), 1995, 36-43.

[52] **E. Visser.** Metaprogramming with Concrete Object Syntax. *Proc. of Generative Programming and Component Engineering (GPCE'02).* LNCS *Vol*. 2487, 2002, 299-315.

[53] **K. De Volder.** Type-Oriented Logic Meta Programming. *PhD thesis, Vrije Universiteit, Brussels, Belgium*, 1998.

[54] **R.J. Walker.** Essential Software Structure through Implicit Context. *Ph.D. dissertation, Department of Computer Science, University of British Columbia*, 2003.

[55] **J. van Wijngaarden.** Code Generation from a Domain Specific Language. *Designing and Implementing Complex Program Transformations. Master's thesis, Universiteit Utrecht, Utrecht, The Netherlands*, 2003.

Received January 2008.

DOI: 10.5755/j01.itc.37.2.11931