INFORMATICA, 2013, Vol. 24, No. 2, 315–337
© 2013 *Vilnius University*

# Equivalent Transformations of Heterogeneous Meta-Programs

Vytautas ŠTUIKYS, Robertas DAMAŠEVIČIUS
*Software Engineering Department, Kaunas University of Technology*
*Studentų 50-415, LT-51368, Kaunas, Lithuania*
*e-mail: vytautas.stuikys@ktu.lt, robertas.damasevicius@ktu.lt*

Received: March 2012; accepted: December 2012

**Abstract.** We consider a generalization of heterogeneous meta-programs by (1) introducing an extra level of abstraction within the meta-program structure, and (2) meta-program transformations. We define basic terms, formalize transformation tasks, consider properties of meta-program transformations and rules to manage complexity through the following transformation processes: (1) reverse transformation, when a correct one-stage meta-program $M^1$ is transformed into the equivalent two-stage meta-meta-program $M^2$; (2) two-stage forward transformations, when $M^2$ is transformed into a set of meta-programs, and each meta-program is transformed into a set of target programs. The results are as follows: (a) formalization of the transformation processes within the heterogeneous meta-programming paradigm; (b) introduction and approval of equivalent transformations of meta-programs into meta-meta-programs and vice versa; (c) introduction of metrics to evaluate complexity of meta-specifications. The results are approved by examples, theoretical reasoning and experiments.

**Keywords:** meta-programming, generalization, transformation, meta-program complexity.

## 1. Introduction and Motivation of the Problem

A meta-program is a program that generates other programs or program parts (Ortiz, 2007). Meta-programming means writing meta-programs. Though meta-programming can be understood and dealt with from different perspectives (e.g., as *frame-based programming* (Cheong and Jarzabek, 1999), *aspect-oriented programming* (Kiczales *et al.*, 1997), *generative programming* (Eisenecker and Czarnecki, 2000), *generic programming* (Reis and Järvi, 2005), *feature-oriented programming* (Trujillo *et al.*, 2007a)), following Veldhuizen (2006) we consider meta-programming as a program generalization and generation technique. We define meta-programming as an algorithmic manipulation of programs as data aiming to support *generative reuse through generalization* (Damaševičius and Štuikys, 2008). The technique enables, at the construction time, to develop a *more abstract* executable specification (*meta-program*) from which programs are generated on demand automatically, at the use stage. Heterogeneous meta-programming is based on using at least two languages for the development of a meta-program. The language at a lower-level of abstraction, called *target language*, serves for expressing domain functionality. A target program written in the target language is used as data to perform manipulations at a higher-level of abstraction. The language at a higher-level of abstraction, called *meta-language*, serves for expressing generalization of a target program through transformations according to the pre-scribed requirements for change.

In this paper, we consider the reverse engineering-based approach to meta-programming when a given meta-program is transformed into a *meta-meta-program* of the same functionality. Why such a transformation is needed? The first and basic reason is complexity. Complexity analysis is an important factor of *software usability* (Sobiesiak and Diao, 2010), because higher complexity usually leads to lower usability and vice versa. As complexity of systems and their components grows continuously, it is commonly agreed that complexity *management* through program transformations and generative reuse is a good (if not the only) solution. Therefore, designers and researchers try to enhance reuse by anticipating possible product changes in advance. That leads to the development of generative components and program generators implemented using meta-programming, i.e., meta-programs.

The second reason is the increase of meta-programs complexity. In such a context, the over-generalization of meta-programs may occur. The better solution is to split the meta-program with the high scope of possible reusability into several parts across different levels of abstraction. In this case, meta-programming may be seen as a tool to manage the design complexity. Finally, the transformation-based view requires a formal description of the process, which then provides a background to automate the process (for practical needs of meta-program transformations, see also Section 3).

From the design perspective, a meta-designer, i.e., a system designer who provides end-users with capabilities of participation in the design process (Fischer and Scharff, 2000), develops meta-meta-programs aiming at managing variations to support a large scale of possible reuse. Each group of variants is specified as a separate meta-program that can be easily derived from the meta-meta-program specification via automatic transformation. Such transformation occurs on demand when the particular needs of the lower-level users (designers, maintainers, etc.) are identified. The delivery process then follows from the meta-designer to the lower-level user. The process enables to disclose only those features of a meta-program that are needed for a concrete user and in such a way as to not reveal the whole meta-meta-program as a valuable intellectual property artefact.

What is needed to support the vision provided above is the introduction of an extra level of abstraction in the meta-program development. In this paper, we address this need as a transformation task when the given meta-program is transformed into a meta-meta-program while preserving the same functionality. Our contribution is: (a) formalization of the transformation processes within heterogeneous meta-programming; (b) introduction and approval of equivalent high-level transformations of meta-programs into meta-meta-programs and vice versa; (c) introduction of metrics (the number of meta-parameters and Cyclomatic Index) to evaluate complexity of meta-programs.

The rest of the paper is organized as follows. Section 2 analyzes related works. Section 3 outlines theoretical backgrounds for heterogeneous meta-programming research. Section 4 provides the definition of basic terms. Section 5 formulates the tasks. Section 6 describes rules to support the meta-programming-based transformation approach. Section 7 describes properties of the transformations. Section 8 provides the experimental approval of the proposed theoretical statements. Section 9 summarizes the results and provides their evaluation. Finally, Section 10 provides conclusions.

## 2. Related Work

Program transformation is a wide topic having applications in many areas of software engineering including compilation, optimization, re-factoring, program synthesis and generation, software renovation and evolution, etc. Here, we restrict ourselves and analyze the most relevant and informative works (i.e., reviews, surveys, if available) in two aspects only: methodological (e.g., program transformation taxonomies) and meta-programming related paradigms (e.g., linguistic aspects of transformations).

Program transformation taxonomies are the results of analysis and classification of program transformations based on a selected number of criteria such as:

(1) The levels of abstraction before and after transformation, and preservation of semantics during transformation (Visser, 2001). The semantics-preserving changes correspond to the well-known concept of software *re-factoring* (Fowler, 1999; Winter, 2004).
(2) The object, method and goal of transformation (the *what-how-why* taxonomy by Winter (2004)).
(3) The dimensions of software change in the context of software evolution and maintenance (Buckley *et al.*, 2003; Benestad *et al.*, 2009).

By comparing taxonomies of Benestad *et al.* (2009), Buckley *et al.* (2003) with the ones of Visser (2001), Winter (2004), we conclude that two terms (i.e., change and transformation) define the same issue but from different perspectives and scope (e.g., development and evolution). The following observation is important to state in this context: the nature of software evolution now is shifting to "*a continuous process, in which there's no neat boundary between development and evolution*" (Boehm, 2010).

Formal and semi-formal description of meta-programs, meta-programming and related higher-level programming methodologies (generic programming, generative programming, aspect-oriented programming, etc.) and transformations for implementing higher-level programs has been intensively studied by many researchers. Intermediate representations, notations and languages such as Abstract Syntax Tree (AST), attribute grammars, rewrite operators, etc. are often introduced and used to simplify the specification of particular sequences and stages of program transformations (Schordan and Quinlan, 2005). Applying transformations to intermediate representations rather than specific domain languages makes transformation tools more reusable. Taha (1999) was the first to provide a formal description for a multi-staged programming language. This theory can be used to prove equivalency between two staged programs, or between a target program and its staged (or meta-) program. Mens *et al.* (2002) consider program transformation formally using a graph-based model. Sheard (2001) reviews and summarizes the accomplishments and research challenges in describing formal meta-programming systems.

In generic programming, transformations are primarily used to instantiate a specifically customized component instance from a generic component. Becker (2000) presents a formal view on a generic component and its customization interface, focuses on external representation and internal realization of the variability of a generic component. Batory (2004) describes formally several transformations used in generic programming such as composition, modularization. Fahmy *et al.* (2001) specify architectural transformations of software at and between different levels of abstraction, including the *lift transformation* that raises elements of a lower-level program to higher levels in the system hierarchy. To some extent, the lift transformation is similar to the reverse transformation described in this paper. Cordy and Sarkar (2004) demonstrate that meta-programs can be derived from higher level specifications using *second order source transformations*. Francis (2004) describes a program transformation tool, called *Metagene*, for generating C++ meta-programs (expressed using template classes) from formal specifications. Reis and Järvi (2005) present a formal model of generic programming based on the category theory and describe formal transformations for developing generic programming algorithms. Trujillo *et al.* (2007b) describe ideas to generate meta-programs from abstract specifications of synthesis paths. The execution of such a meta-program code synthesizes a target program of a product line. In the context of aspect-oriented programming, Yang (2009) presents a formal analysis of aspect weaving for introducing code modifications in components through aspects.

In feature modelling, especially for product line engineering, formal models of product features and different interactions between them are important for further implementation of meta-programs or software generators implementing product lines. Janota and Kiniry (2007) define a formalized feature meta-model to support reasoning about and within feature model approaches, feature models, and feature trees and their configurations. Westfechtel and Conradi (2009) describe a formal description of multi-variant models in the context of product line engineering, describe transformation processes on such models including editing and product configuration, and discuss the construction and representation of models incorporating multiple variants. Ebraert *et al.* (2009) present a formal model of change-oriented programming based on feature diagrams, in which features are seen as sets of changes (or high-level transformations) that can be applied to a base program.

Though there are many slightly different views on meta-programming as a sub-field of program changeability and transformation, meta-programming is not only a topic for academic research (this view might come to one's mind due to the restricted analysis and because of our components to be considered later are small and specific). There are also announcements on industrial (i.e., large-scale) applications of the meta-programming-based systems (e.g., *frame-based programming* (Bassett, 1997), *XVCL-based programming* (Jarzabek and Pettersson, 2006), *template meta-programming* (Karaila and Systa, 2007), *pre-processing programming* (Vidacs, 2009)). For example, Jarzabek and Pettersson (2006) evaluate the benefits of generic design via parameterization (which can be seen as another definition of meta-programming) as follows: ease of reuse with adaptation, the overall reduction of conceptual complexity and size of the solution, improved traceability and changeability, which outweigh the cost of the added complexity and lower understandability. Note that Bassett's frame-based and XVCL commands have some conceptual resemblance to our Open PROMOL functions (Štuikys *et al.*, 2002).

Transformation of meta-programs leads to the problem of measuring their semantic equivalence. Software complexity measures can be used to reason about program and meta-program structure and functionality as well as for comparing and evaluating meta-programs; see Damaševičius and Štuikys (2010). Though the researchers acknowledge the importance of dealing with various aspects of transformation-based approaches within the meta-programming paradigm, complexity evaluation of meta-programs only recently attracted the attention of researchers. For example, Ross (2006) presents an analysis based on cost-bound functions and abstract-interpretation approximation of program states for C++ generic programs (templates) aiming to determine the best set of template parameters (types) for optimal performance. Pataki *et al.* (2006) propose a multi-paradigm complexity metric based on McCabe's cyclomatic complexity to evaluate structural complexity of aspect-oriented programs written in AspectJ.

## 3. Theoretical Backgrounds and Motivation

In this paper, we consider transformations of meta-programs that are designed using heterogeneous meta-programming techniques. Such techniques can support generalization in software development when the main focus is given to automatically creating programs, which are derived from the meta-program specification. To express generalization, at least two languages, i.e., meta- and target ones, are used in heterogeneous meta-programming (in the simplest case).

A target language (TL) can be used to express multiple aspects such as *representation* (e.g., of mathematical equations, plain text), *security aspects* (e.g., in web-based applications), *computational* (i.e., the conventional use), *various domain models* and *distribution aspects of web systems* (Damaševičius *et al.*, 2004). The latter requires the use of more than two languages.

The role of a meta-language (ML) is to express one- or two-level generalizations through various manipulations over TL programs. Manipulations are described using ML constructs in *the structural programming manner* according to pre-scribed requirements for change that are to be anticipated in advance, e.g., at the domain analysis phase. Various languages can be used in the role of a ML: from dedicated (e.g., Open PROMOL (Štuikys *et al.*, 2002), MetaL), domain-specific (e.g., Perl, PHP, ASP) or general purpose programming language (e.g., C++, Java, Visual Basic). In the latter case, only a part of language capabilities is used to express the generalizations.

Using two languages, it is possible to realize, e.g., two-level generalization, i.e., to develop meta-meta-programs. Scope of generalization can be further extended if there is a need to use more than two languages. Further in this paper, by generalization we mean the introduction of an extra level in the meta-program structure that leads to the transformation of a meta-program into a meta-meta-program.

In this context, however, we need to motivate the practical value of two-level generalization and adequate meta-program transformations. Though we have identified the possibility of two-level generalization much earlier (Damaševičius, 2005), this idea can also be tracked from Taha (1999), only recently such an approach has been practically implemented in the development of three web-based components for real-world applications (i.e., web sites) using two-stage meta-programming (Montvilas, 2009; Štuikys *et al.*, 2009). Our recent experiments with embedded software components, which are implemented using one-stage or two-stage meta-programming due to the use of multiple criteria (e.g., energy, performance, accuracy, memory requirements and their various trade-offs) showed the benefits of the approach. Finally, we have been convinced in the soundness of the approach for generating learning objects (e-learning domain) from the multi-staged generative learning objects.

A kind of meta-program transformations analyzed in this paper is, in fact, a *partial evaluation* of meta-programs. Partial evaluation (Jones *et al.*, 1993), also called partial deduction or program specialization, is an automatic program transformation technique that aims for the specialization of programs, with regard to parts of their input, while preserving program semantics (Iranzo, 2003). The algorithm of partial evaluation is encoded at the meta-meta-level. While the instantiation of a meta-program (i.e., generation of an instance through forward transformation) can be considered as a full evaluation of a meta-program with respect to its meta-parameter values (i.e., substitution of meta-parameters with their values and execution of a transformation algorithm at the meta-level), a partial evaluation of a meta-program concerns only a subset of the meta-parameters. The result is another meta-program in case of the *early* (implicit, online) partial evaluation, when the values of the evaluated meta-parameters are known in advance and a meta-meta-program in case of the *late* (explicit, offline) partial evaluation, when the partial evaluation is encoded as a meta-meta-program. As was proven by Welinder (1996), two expressions (programs) are equivalent if they, when evaluated in the identical environments, always produce identical results. In case of meta-programming, two meta-programs are semantically equivalent if they generate identical sets of instances (programs) for the same values of meta-parameters.

It should be emphasized that this kind of transformations are purposeful if two conditions are satisfied: (a) there is a great variability in the domain (in terms of meta-programming this means a great number of parameters and their values); (b) this variability is known either in advance or it can be extracted from domain (e.g., through domain modelling). It is difficult to perform two-level generalization in a straightforward manner. To accomplish the task the best strategy is to start from the single-level meta-program, and then proceed to the second level through some kind of transformations. As the complexity of generalization and transformations involved is indeed challenging, we are not able to consider all aspects of the problem in detail here. We restrict ourselves with the tasks as they are defined and stated in Sections 4 and 5.

## 4. Definitions of Basic Terms

*Nomenclature:*

$L_M, L_T$ – a formal notation of a meta-language (ML) and target language (TL), respectively;

$M^0, M^1, M^2$ – program representation (i.e., program, meta-program and meta-meta-program) at different levels;

$\mu^0, \mu^1, \mu^2$ – model representation (i.e., models of a program, meta-program and meta-meta-program) at different levels;

$M_I^1, M_B^1$ – meta-interface and meta-body of a meta-program as a full specification, respectively;

$M_I^2, M_B^2$ – meta-meta-interface and meta-meta-body of a meta-meta-program as a full specification, respectively;

$\mu(M_I), \mu(M_B)$ – model of a meta-interface and meta-body at any level, respectively;

FT – Forward Transformation (notation used in text);

$F^1,\ F^2$ – first and second stage of forward transformations, respectively.

RT – Reverse Transformation (notation used in text);

$R^1,\ R^2$ – first and second stage of reverse transformations, respectively.

$\xrightarrow{x}$ – Forward Transformation dependent on data $x$ (notation used in formulae);

$\xleftarrow{x}$ – Reverse Transformation dependent on data $x$ (notation used in formulae);

$S$ – the full space of meta-parameters including parameter names and their values;

$p_i, V^i$ – a parameter name and a set of values of the parameter, respectively;

$A^0, A^1, A^2$ – a program in TL ($|A^0| = 1$), a set of programs derived from $M^1$, and a set of meta-programs $M_i^1$ derived from $M^2$ (via the first stage of FT), respectively;

$A_i^1$ – a subset of programs derived from the set $M_i^1$ (through the second stage of FT), ($i = [1; q]$, $q$ – the number of meta-programs).

**Note.** By the word "program" we mean a concrete program as an instance written in TL (formal notation $L_T$) throughout the paper.

Other notations are introduced within the paper.

For simplicity reasons, in Sections 4–6, we refer to heterogeneous meta-programming that uses only two languages (i.e., the meta and target ones). Below we present definitions of the basic terms that correspond to the simplest understanding of heterogeneous meta-programming (i.e., the use of two languages only). Elsewhere, when we refer to heterogeneous meta-programs, the word 'heterogeneous' is omitted for short.

The other observation relates to the selection of languages to support meta-programming. In general, the selection of a TL depends usually on the application domain, while the choice of a ML may be either optional (Štuikys and Damaševičius, 2003) or that selection may be pre-specified by the other criteria such as requirements of the system, relevance to the use of TL or even relevance to the meta-designer's flavor. To illustrate the basic concepts of the approach below, we use Open PROMOL as a ML (Štuikys *et al.*, 2002) and simple text strings as a TL.

```
     $
a)       “Select a number of inputs”         {2..10}  n:= 3;
         “Select a type of operation”        {Λ, V}   p:= Λ;
     $
         @- here is a comment; the next line is meta-body
         Y = X1 @for[i, 2, n, { @sub[p]  X@sub[i]}];
---------------------------------------------------------------
b)             Y = X1 Λ X2 Λ X3;
```

Fig. 1. Illustrative example: meta-program (a) and its derivative instance (b), i.e., target program.

DEFINITION 1. Meta-programming is an algorithmic manipulation of programs as data aiming to support generative reuse through generalization. Generalization is parameterization (at one or two stages) specified by constructs of a ML.

DEFINITION 2. Meta-program is a higher-level executable specification (aka meta-specification), which is coded using two languages, $L_M$ and $L_T$, where $L_M$ is a ML, and $L_T$ is a TL. Meta-program specifies a set of programs in $L_T$. The programs are derived from the meta-program when it is executed.

DEFINITION 3. Meta-meta-program is a meta-program in which the generalization is presented at two levels. Some generalization aspects are presented at the meta-meta level using $L_M$, while the rest part of generalization is presented at the meta-level using $L_M$ and $L_T$. When executed, the meta-meta-program produces a set of meta-programs.

DEFINITION 4. Meta-program's structural model $\mu^1$ is a composition (denoted as "$+$" in (1) of two interrelated parts, i.e., meta-interface *model,* and meta-body *model*:

$$\mu^1 = \mu(M_I^1)+\mu(M_B^1). \tag{1}$$

EXAMPLE 1. In Fig. 1a, we present the implementation of $\mu^1$, which specifies the task of generating homogeneous Boolean logic equations of any length. Here the equations are treated as target programs. The meta-program is described using Open PROMOL functions as $L_M$ and a text string is specified as $L_T$. Meta-interface $M_I^1$ is presented between symbols '\$'; $n, p$ – are meta-parameters (shortly parameters); their values are given within braces. Meta-body $M_B^1$ is presented after the second symbol '\$'. $M_B^1$ is a composition of a target program (see uppercase) and meta-functions (@**for**[...] – loop function; $i$ – loop variable; @**sub**[...] – parameter substitution by its value). In Fig. 1b, we present a generated instance in $L_T$ derived from the given meta-program.

DEFINITION 5. Formally, meta-interface model of a given meta-program $M^1$ is a $n$-dimensional (meta-) parameter space $S$ ($S \in M^1$):

$$\mu(M_I^1) = S, \tag{2}$$

where

$$S = (S_1, S_2, \dots, S_n), \tag{2.1}$$

$$S_i = \big(p_i, V^i\big), \quad S_i \subset S, \quad \text{where } i \in [1;n],\ n = |P|;\ P = \cup p_i, \tag{2.2}$$

$$V^i = \big(v_1^i, v_2^i, \dots, v_{k_i}^i\big), \tag{2.3}$$

where $n$ – the number of parameters; $p_i$ – the parameter name (identifier); $P$ – a set of parameters; $V^i$ – a set of values of the parameter $p_i$ and $v_{j_0}^i \in V^i$ $(1 \leqslant j_0 \leqslant k_i)$ is an initial or default parameter value (see Fig. 1a).

DEFINITION 5a. Meta-interface $M_I^1$ is the implementation of its model $\mu(M_I^1)$using $L_M$.

PREMISE. This paper considers meta-programming-based transformations at the meta-design stage under the following pre-conditions: (a) meta-parameter space is pre-specified in advance and can not be changed; (b) transfer of meta-parameters from the meta-parameter space of $\mu(M_I^1)$ to the meta-meta-parameter space of $\mu(M_I^2)$ is the only permissible source of meta-meta-parameters for $M^2$.

DEFINITION 6. Meta-body $M_B^1$ is a part of $M^1$ that specifies an algorithm of anticipated modifications using two languages $L_M$ and $L_T$. Model of $M_B^1$ is a union of finite strings $\lambda_j^*$ (i.e., $0 < j < J$); $J$ – number of strings;

$$\mu(M_B^1) = \bigcup_j \lambda_j^*, \tag{3}$$

where $\lambda_j^* \in T^* \cup N^*$ and $T^*, N^*$ are generalized terminal symbols and non-terminal symbols of a given target program to be generalized, respectively; with asterisk (*) we specify a generalization output as a result of manipulation with a target program throughout the paper. As not all terminal and non-terminal symbols have to be modified via generalization, we identify $T^*,\ N^*$ using (4) and (5):

$$T^* = T_1(L_T) \cup T_2^*(L_T), \tag{4}$$

$$N^* = N_1(L_T) \cup N_2^*(L_T). \tag{5}$$

Furthermore, $T = T_1 \cup T_2;\ N = N_1 \cup N_2$; where $T$ and $N$ are terminal and non-terminal symbols of the target program given in $L_T$ before generalization, respectively; $T_1, N_1$ are terminal and non-terminal symbols that are not modified via generalization; $T_2, N_2$ are terminal and non-terminal symbols that are to be generalized, i.e., modified according to the given requirements ($T_1 \cap T_2 = \varnothing$ and $N_1 \cap N_2 = \varnothing$). Also we assume that $T_2 \neq \varnothing$ and $N_2 \neq \varnothing$.

Terminal symbols $T_2$ are to be transformed into $T_2^*$ and non-terminal symbols $N_2$ are to be transformed into $N_2^*$ according to transformation rules (6) and (7), respectively:

$$T_2^* \xleftarrow{\lambda(L_M), R(G)} T_2, \tag{6}$$

$$N_2^* \xleftarrow{\lambda(L_M),R(G)} N_2, \tag{7}$$

where $\lambda(L_M)$ are strings of the meta-language $L_M$used for generalization (change); $R(G)$ – requirements for generalization $G$; and "$\overset{x}{\leftarrow}$" denotes the reverse transformation rules dependent on data $x$, which will be specified in more detail in Section 6.

Definition 7. Meta-meta-program's structural model (denoted as $\mu^2$) is a composition (denoted by the symbol '+') of three interrelated models as it is identified by (8):

$$\mu^2 = \mu(M_I^2) + (\mu'(M_I^1) + \mu(M_B^1)), \tag{8}$$

where $\mu(M_I^2)$ is the meta-meta-interface model and $(\mu'(M_I^1) + \mu(M_B^1))$ is the model of the meta-meta-body and $\mu'(M_I^1), \mu(M_B^1)$ are models of $M_I^1$ and $M_B^1$, respectively.

Definition 8. Reverse transformation (RT) is a meta-program (Case 1) or meta-meta-program development process (Case 2) that is performed through generification of a program or a meta-program aiming to develop a meta-program or a meta-meta-program, respectively.

In Case 1, RT (formally, $R^1$) is performed through modifications of a program model (see also (4)–(7)) into a meta-program model (see Definition 4 and (1)) according to requirements for change/generalization, which are expressed through the ML constructs.

In Case 2, RT (formally, $R^2$) is a process of changing the meta-program structure by re-factoring its structural model into the meta-meta-program model (see Definition 7 and (8)) through the introduction of the extra generalization level but preserving the same parameter space $S$ (see (2) and (2.1)).

Definition 9. Forward transformation (FT) of a meta-program into a set of (target) programs is a process of identifying the pre-defined meta-parameter values, and then according to those values, deriving programs through the generation process automatically. Meta-programming-based program generation is a FT process.

Definition 10. One-stage FT (formal notation $F^1$) of a source meta-meta-program into a set of target meta-programs is a process of identifying the pre-defined meta-meta-parameter values, and then, according to those values, deriving meta-programs in the same way as generating instances (see Definition 9).

Definition 11. Two-stage FT ($F^2$) of a meta-meta-program into a set of programs is a process that consists of two consecutive one-stage FTs, when, first, a set of meta-programs are generated and then, in the second stage, programs are derived from each meta-program according to the pre-scribed meta-parameter values.

Definition 12. Cyclomatic Index (CI) of a meta-program is the total number of programs that can be derived from the meta-program via the FT process. Cyclomatic Index (CI) of a meta-meta-program is the total number of different meta-programs that can be derived from the meta-meta-program via the one-stage FT process.

Formally, CI is computed by enumerating all possible different paths within the meta-program execution process graph, where the initial node is the first statement of the meta-program and the ending node is the last statement, when a target program is produced as a result of the process. CI enables to compare and evaluate the complexity of meta-programs of the same or related functionality. For example, if two meta-programs (having $\mathrm{CI}_1$ and $\mathrm{CI}_2$, respectively) are derivatives of the base meta-meta-program with $\mathrm{CI}_m$, then relations ($<$, $>$, $\ll$, $\gg$, etc.) among entities ($\mathrm{CI}_1$, $\mathrm{CI}_2$, and $\mathrm{CI}_m$) enable to reason about the meta-(meta-)program complexity and to evaluate it. In general, the need for such measures is described in Oram and Wilson (2010). They have both theoretical and practical importance due to ever-increasing complexity of systems, the software content growth and the shift to higher and higher abstraction levels in designing systems. In the context of this paper, such measures are important for comparing, evaluating and managing complexity of meta-specifications used in different applications (apart of the application (see also Section 8.2 and Table 1), we have also identified the problem dealing with the other applications, i.e., generative learning objects and web-based meta-meta-components).

Note that Definition 12 is the application of the program Cyclomatic Complexity Index (McCabe, 1976) to meta-programming to measure meta-(meta-)program complexity.

DEFINITION 13. Semantics of a program is its functionality expressed through the encoded algorithm. By analogy, semantics of a meta-program is its functionality expressed through algorithmic manipulations by meta-constructs applied on the base target program within the given space of meta-parameters. Semantics of a meta-meta-program is its functionality expressed through algorithmic manipulations by meta-constructs applied on the base meta-program within the given space of meta-meta-parameters.

DEFINITION 14. A meta-parameter is said to be *active* if it performs the prescribed role for change. A meta-construct of the ML is *active* if its parameter (or parameters) is (are) active. *Deactivation* is a process to change the role from *active* (initially prescribed to the meta-parameter and the meta-construct manipulating on the meta-parameter) to *passive* (when the meta-parameters and meta-constructs are treated as entities of a TL).

Note that deactivation is a mechanism to manage transformations (e.g., when $M^1$ is transformed into $M^2$). ML should provide such a mechanism (it is the symbol "\", see Fig. 3).

## 5. Transformation Tasks

**Task 1**. Given: (a) a correct source meta-program $M^1$coded in $L_M$ and $L_T$, (b) model of $M^1$ (i.e., $\mu^1(M^1)$ that is presented as (1), (2) and (3) (see Definitions 4–6) and $M_I^1$ is separable from the meta-body $M_B^1$). The task is to transform the meta-program $M^1$ into a set of programs A (coded in $L_T$) through the one-stage FT process $F^1$.

With regard to the set A, there might be 3 useful transformation cases:

(1) $\mathcal{A} = A^1$ (i.e., all possible instances are derived from $M^1$ based on $n$-dimensional parameter space $S$ (see Definitions 5 and 12 – only its first part)),

(2) $\mathcal{A} \subset A^1$ (i.e., a prescribed subset of instances is derived from $M^1$ through transformation);

(3) $\mathcal{A} = A^0(|A^0| = 1,\ A^0 \in A^1)$, (i.e., a concrete program is generated through the transformation, which is specified by the default parameter values $(\forall i(v^i_{j_0} \in V^i),\ (1 \leqslant j_0 \leqslant k_i$; see Definition 5 and Fig. 1a, for concrete details).

The choice of a variant depends on the user's needs and capabilities (modes) of the $L_M$ processor that performs the transformation. If we assume that $M^1$ is syntactically and semantically correct, Task 1 can be automatically solved through the FT process that is supported by $L_M$ processor. The only thing the user needs to do is to select the variant by specifying the parameter values. Figure 1 presents the solution for case 3 using a simple illustrative example.

**Task 2.** Given: (a) and (b) as in Task 1, transform $M^1$ into $M^2$ (through the RT process) so that the following conditions are satisfied: (a) parameter space $S$ is decomposed into two subspaces, i.e., $S = (S_{M^2} \cup S_{M^1}),\ (S_{M^2} \cap S_{M^1} = \varnothing;\ S_{M^2} \neq \varnothing;\ |S| \geqslant 2)$, where $S_{M^2}$ and $S_{M^1}$ are subspaces of the meta-meta-level and meta-level parameters, respectively; (b) model of $M^2$, i.e., $\mu^2(M^2)$ is specified by (8).

**Task 3.** Given a meta-meta-program $M^2$ derived from its source meta-program $M^1$ through the RT process, perform the two-stage FT as follows: in the first stage, transform $M^2$ into a set of meta-programs $A^2$, where $A^2 = \bigcup_{i=1}^{q} M^1_i$ ($q$ – the number of meta-programs, $M^1_i$ – a concrete meta-program); and in the second stage, for each $i$, transform $M^1_i$ into $A^1_i$, where $A^1_i$ is a subset of programs derived from $M^1_i$.

EXAMPLE 2. To illustrate the formation of $M^2$ a more complex meta-program $M^1$ is needed. The extended meta-program (see Fig. 2) is a modification of the previous one (see Fig. 1) by introducing new features (i.e., the possible change of the function name (parameter $s1$), the argument name (parameter $s2$), the second equation is added with the different number of arguments (parameter $n2$) and operation type (parameter $p2$)).

```
        $
          "Select a notation symbol for functions"                {Y, Z, S}   s1:= Z;
          "Select notation symbol for arguments"                  {X, A, B}   s2:= A;
          "Select a number of inputs for the first equation"      {2..10}     n1:= 3;
(a)       "Select a type of operation for the first equation"     {Λ, V}      p1:= V;
          "Select a number of inputs for the second equation"     {2..10}     n2:= 4;
          "Select a type of operation for the second equation"    {Λ, V}      p2:= Λ;
        $
          @sub[s1]1 =  @sub[s2]1 @for [i, 2, n1, { @sub[p1] @sub[s2]@sub[i]}];
          @sub[s1]2 =  @sub[s2]1 @for [i, 2, n2, { @sub[p2] @sub[s2]@sub[i]}];

(b)       Z1 = A1 V A2 V A3;
          Z2 = A1 Λ A2 Λ A3 Λ A4;
```

Fig. 2. An extended meta-program (a) and its derivative instance (b) (see also Fig. 1).

```
      @- this is the beginning of meta-meta-interface
      $
(a)   "Select a notation symbol for functions"           {Y, Z, S}    s1:= Z;
      "Select a notation symbol for arguments"           {X, A, B}    s2:= A;
      $
      @- the beginning of meta-meta-body
        @sub [{
            $
        "Select a number of inputs for the first equation"     {2..10\}   n1:= 3;
        "Select a type of operation for the first equation"    {Λ, V\}    p1:= V;
        "Select a number of inputs for the second equation"    {2..10\}   n2:= 4;
        "Select a type of operation for the second equation"   {Λ, V\}    p2:= Λ;
            $
                }]
      @sub[s1]1 =  @sub[s2]1\@for[i, 2, n1, { \@sub[p1] @sub[s2]\@sub[i]}];
      @sub[s1]2 =  @sub[s2]1\@for[i, 2, n2, { \@sub[p2] @sub[s2]\@sub[i]}];
```

```
        $
        "Select a number of inputs for the first equation"     {2..10}    n1:= 3;
(b)     "Select a type of operation for the first equation"    {Λ, V}     p1:= V;
        "Select a number of inputs for the second equation"    {2..10}    n2:= 4;
        "Select a type of operation for the second equation"   {Λ, V}     p2:= Λ;
        $
        Z1 =  A1 @for [i, 2, n1, { @sub[p1] A@sub[i]}];
        Z2 =  A1 @for [i, 2, n2, { @sub[p2] A@sub[i]}];
```

```
(c)     Z1 = A1 V A2 V A3;
        Z2 = A1 Λ A2 Λ A3 Λ A4;
```

Fig. 3. Meta-meta-program (a), its derivative meta-program (b) derived using the pre-defined meta-meta-parameter values and instance (c) derived using the pre-defined meta-parameter values.

EXAMPLE 3. This example explains a result of RT of $M^1$ (see Fig. 2) into $M^2$ (see Fig. 3). $M_I^2$ has meta-meta-parameters $s1, s2$. The symbol "\" is used to change the role of the ML constructs to the TL symbol, e.g., to deny the role of enclosing braces as ML symbols where it is needed (see Fig. 3, **@sub** is the function that returns a value of its argument).

## 6. Transformation Method

We describe the method of transforming $M^1$ into $M^2$ (Task 2) as a sequence of actions supplemented by a set of transformation rules. The actions are performed under the following initial conditions: (1) the given meta-program $M^1$ is correct, i.e., its syntax and semantics is described by Definitions 2, 4–6; (2) the parameter subspace $S_{M^2}$ is specified in advance.

*Step* 1. Check eligibility of meta-parameters within $S_{M^2}$ (Rules 1 and 2). If dependable parameters appear in different subspaces $S_{M^2}$ and $S_{M^1}$, those parameters should be moved either to $S_{M^2}$ or to $S_{M^1}$ (this is the correction of given requirements; $S^*_{M^2}$, $S^*_{M^1}$ – are the corrected subspaces).

*Step* 2. Select the structure of $M^2$, i.e., identify models for meta-meta-interface $M_I^2$ and meta-meta-body $M_B^2$ (initially $M_I^2$ is empty).

*Step* 3. Fill $M_I^2$ by $S_{M^2}$ or by $S^*_{M^2}$ (if a correction in *Step* 1 was made).

*Step* 4. Deactivate meta-parameters $S_{M^1}(S^*_{M^1})$ within the meta-meta-body $M_B^2$ (see Definition 14 and Fig. 3).

*Step* 5. Deactivate meta-constructs within the meta-meta-body $M_B^2$, which relate to the deactivated meta-parameters (see Fig. 3).

The rules we present below with examples are based on the definitions and models (see Section 5). The solution of Tasks 1 and 3 is the one-stage or two-stage FT process, i.e., $F^1$ and $F^2$, respectively: it is performed automatically using a ML processor, if the adequate meta-specification is already developed. The solution of Task 2 is about the development of meta-specification as an input of Task 3. The rules describe how a meta-program is to be developed.

**Rule 1.** If the meta-interface of the given meta-program has *no dependent parameters* and there are *no specific requirements* to form the meta-meta-level, then the space $S$ can be decomposed into two subspaces arbitrarily, i.e., $S = (S_{M^2} \cup S_{M^1}), (S_{M^2} \cap S_{M^1} = \varnothing;\ S_{M^2} \neq \varnothing;\ |S| \geqslant 2)$.

If there are specific requirements specified by a meta-designer to form the meta-meta-level, then the parameter space $S$ is to be decomposed into two subspaces (i.e., $S_{M^2}$ and $S_{M^1}$) according to the requirements for the semantics of the transformation.

**Note.** As $S_i = (p_i, V^i)$ (see (2.2)) the decomposition is applied to meta-parameters $p_i$ only, i.e., $S_i$ is treated as an element of $S$.

**Rule 2.** If the meta-interface of a meta-program has dependent parameters (e.g., there are mutual exclusive parameter values, or some parameter requires a specific value of another parameter, etc.) those parameters are to be placed at the *same level* (i.e., meta or meta-meta).

EXAMPLE 4. The example is a modification by introducing dependable parameters $s1$ and $s2$ in the example of Fig. 3 (see Fig. 4a; here 'neq' and 'eq' are Open PROMOL's operations 'not equal' and 'equal' for strings, respectively).

**Rule 3.** Given the decomposition of the parameter space $S$ into two subspaces, i.e., $S = (S_{M^2} \cup S_{M^1}), (S_{M^2} \cap S_{M^1} = \varnothing;\ S_{M^2} \neq \varnothing;\ |S| \geqslant 2)$. The transformed meta-body of the meta-meta-program is identified as a concatenation (denoted by '$+$') of two items, i.e., $M_B^2 = (M_I^1(S_{M^1}) + M_B^1)$, where $M_I^1(S_{M^1})$ is a part of meta-meta-body, which serves for specifying the meta-level interface; $M_B^1$ is a part of meta-meta-body, which serves for specifying a meta-body as a set of instances (programs) at the meta-level.

```
a)      $
          "Select a notation symbol for functions"                 {X, Y, Z, S}  s1:= Z;
         [s1 neq {X}] "Select a notation symbol for arguments"    {X, A, B}      s2:= X;
         [s1 eq {X}]  "Select a notation symbol for arguments"    {A, B}         s2:= A;
        $
         @sub[{
         $
         "Select a number of inputs for the first equation"      {2..10\}  n1:= 3;
         "Select a type of operation for first equation"         {Λ, V\}   p1:= V;
         "Select a number of inputs for the second equation"     {2..10\}  n2:= 4;
         "Select a type of operation for the second equation"     Λ, V\}   p2:= Λ;

         $
               }]
         @sub[s1]1 = @sub[s2]1 \@for [i, 2, n1, { \@sub[p1] @sub[s2]\@sub[i]\}];
         @sub[s1]2 = @sub[s2]1 \@for [i, 2, n2, { \@sub[p2] @sub[s2]\@sub[i]\}];
```

```
         $
         "Select a number of inputs for the first equation"      {2..10\}  n1:= 3;
b)       "Select a type of operation for the first equation"     {Λ, V\}   p1:= V;
         "Select a number of inputs for the second equation"     {2..10\}  n2:= 4;
         "Select a type of operation for the second equation"    {Λ, V\}   p2:= Λ;
         $
         Z1 =  X1 @for [i, 2, n1, { @sub[p1] X@sub[i]}];
         Z2 =  X1 @for [i, 2, n2, { @sub[p2] X@sub[i]}];
```

```
c)       Z1 = X1 V X2 V X3;
         Z2 = X1 Λ X2 Λ X3 Λ X4;
```

Fig. 4. Meta-meta-program (a) for the task of Fig. 3, meta-program (b) and its generated instance (c).

## 7. Properties

**Property 1.** Cyclomatic Index $\mathrm{CI}_{M^1}$ (aka $|A^1|$) of a meta-program $M^1$ depends on the characteristics of meta-interface only; the index is independent upon the length (size) of a target program which is derived from the meta-program. The index is calculated using relationship (9) (see also Definition 5 and (2)–(2.3)):

$$\mathrm{CI}_{M^1} \leqslant |V^1| \times |V^2| \times \cdots \times |V^n|. \tag{9}$$

Note that formula (9) specifies the upper bound of the Cyclomatic Index. The inequality identifies the case when some parameters are dependable. In Fig. 4, e.g., parameters $s1$ and $s2$ are dependable because they have a common value (see the implementation of the meta-meta-interface in Fig. 4). The equality sign identifies the case when all parameters are orthogonal.

**Property 2.** Let we have three meta-specifications as follows: (i) $M^1$, (ii) $M^2$ that is derived from the first as a result of one-stage RT and (iii) $M_i^1$, which is a set of meta-programs derived from $M^2$ as a result of the FT process. Then the Cyclomatic Index of $M^2$ is defined by (10), and the number of all programs derived from $M^2$ is calculated by (11):

$$\mathrm{CI}_{M^2} = |A^2| = q, \tag{10}$$

$$|A^1| = \sum_{i=1}^{q} \mathrm{CI}_{M_i^1}, \tag{11}$$

where $|A^1|$ is the number of all programs derived from meta-meta-program $M^2$ through the subsequent transformations $F^2$ and $F^1$ (also $|A^1| = \mathrm{CI}_{M^1}$); and $\mathrm{CI}_{M_i^1}$ is the Cyclomatic Index of the meta-program $M_i^1$.

**Property 3.** All parameters and meta-constructs are active within a meta-program $M^1$. The deactivation process, if implemented correctly with respect to the given subspace $S_{M^2}$, does not change the overall functionality of meta-meta-program $M^2$, but rather performs the partitioning of the functionality between two levels (meta-meta and meta).

It is obvious for the given concrete subspace $S_{M^2}$ and due to the meta-meta-program semantics definition (see Definition 13). As a result, the following corollary is formulated.

**Corollary**. Change of a meta-program $M^1$ within the given parameter space $S$ into a meta-meta-program $M^2$ via the RT process implemented as *Steps* 1–5 (see Section 6) is a semantics-preserving transformation (with regard to the definitions of meta-program and meta-meta-program semantics, see Definitions 13 and 14).

**Property 4.** Let we have the meta-parameter space $S$ as follows: $S = (S_{M^2} \cup S_{M^1}), (S_{M^2} \cap S_{M^1} = \varnothing;\ S_{M^2} \neq \varnothing;\ |S| \geqslant 2)$. Then the number of all possible reverse transformations $R^2$ of a given $M^1$ into $M^2$ is equal or less to the number of all possible permutations of $M^1$ meta-parameters between meta- and meta-meta-levels (except marginal cases when $M^1$ and $M^2$ parameter spaces are empty) as follows:

$$\sigma(R^2) \leqslant 2^{|S|} - 2, \quad 1 \leqslant |S_{M^2}| < |S|. \tag{12}$$

The equality in (12) corresponds to the case when all parameters within the space $S$ are orthogonal. This property follows from Rule 1. Otherwise, this number is less than the right side expression due to Rule 2.

Are all possible transforms from $M^1$ into $M^2$ equivalent within the given meta-parameter space? We argue that all possible transforms that satisfy the prescribed properties and rules are equivalent. To approve the later statement, we (1) have generated all possible transforms and checked their equivalency experimentally using a selected example (Section 8.1); (2) we have presented experiments we carried out with meta-meta-programs and meta-programs for real world tasks (Section 8.2).

## 8. Experiments and Case Study

### 8.1. *Generation of Bit Strings Using Meta-Programming*

An example of bit string generation using Open PROMOL as a meta-language is given in Fig. 5. Note that bit strings can be considered as low-level representation of computer

```
$
"Enter the value of the 1st bit" {0,1} s1:=0;
"Enter the value of the 2nd bit" {0,1} s2:=0;
"Enter the value of the 3rd bit" {0,1} s3:=0;
$
@sub[s1]@sub[s2]@sub[s3]
```

**(a)**

```
$
"Enter the value of the 1st bit" {0,1} s1:=0;
$
$
"Enter the value of the 2nd bit"{0,1} s2:=0;
"Enter the value of the 3rd bit"{0,1} s3:=0;
$
@sub[s1]\@sub[s2]\@sub[s3]
```

**(b)**

```
$
"Enter the value of the 2nd bit" {0,1} s2:=0;
"Enter the value of the 3rd bit" {0,1} s3:=0;
$
0@sub[s2]@sub[s3]
```

**(c)**

```
000 001 010 011 100 101 110 111

000 001 100 101 010 011 110 111

000 010 100 110 001 011 101 111
```

**(d)**

Fig. 5. Example of bit string meta-programming.

programs using a 2-symbol alphabet $A = \{0, 1\}$. Each bit string, in fact, can represent a different component of a computer program. In Fig. 5, an original meta-program $M^1$ for generation of 3-length bit strings is presented. In Fig. 5b, an example of an equivalent variant of meta-meta-program $M^2$ derived from $M^1$ via RT for a meta-parameter $s1$ moved to the meta-meta-level is presented. In Fig. 5c, an example of the variant of meta-program $M^1$ generated from $M^2$. Finally, in Fig. 5d, examples of different variants of bit strings that can be generated from $M^1$ via one-stage FT or from $M^2$ via two-stage FT. Note that though the instances of bit strings in Fig. 5d can be generated in a different order, depending upon the placement of meta-parameters at different levels of meta-meta-program (Fig. 5b demonstrates only one variant of such placement, when meta-parameter $s1$ is moved to the meta-meta-level; another variants include lifting meta-parameters $s2$ or $s3$), the sets of generated bit strings are equal. Therefore, the transformations leading to the generation of such bit strings are equivalent.

### 8.2. *Development and Complexity Analysis of Meta-Programs in Embedded Hardware Domain*

To be aware about the validation of the approach for more complex tasks, we also carried out some experiments (by reproducing former experiments for a real task in the context of this paper) as a case study below. We developed meta-programs that generate fault-tolerant Intellectual Property components (soft IPs) for fault-tolerant embedded hardware that relies on the concept of *redundancy* (Johnson, 1989), i.e., the addition of extra hardware resources (*space redundancy*), additional time to perform system functions (*time redundancy*), or addition of redundant data to ensure reliability during a transfer of data via system interconnections (*data redundancy*), as compared to what is needed for normal system operations.

We treat the soft IP (Štuikys and Damaševičius, 2003), whose reliability we increase, as a black-box component. This soft IP is wrapped with a trivial circuitry that performs majority voting of redundant signals in search of a possible error. To perform experiments and evaluate the models, we have transformed the previously developed meta-program (see Fig. 6 and paper Štuikys and Damaševičius, 2003) into the meta-meta-program (see Fig. 7).

```
$
       "Select a component:
            1 - dragonfly core
            2 - i8051 microcontroller
            3 - Free6502 core"                       {1,2,3}       c := 1;
[c = 1] "Enter address bus width for program memory" {8,16}        prgm := 16;
[c = 1] "Enter address bus width for data memory"    {4,8}         dm := 8;
[c = 1] "Enter address bus width for i/o"            {4,8}         io := 8;
[c = 3] "Generate Free6502 core version for:
            1 - synthesis
            2 - debugging"                           {1,2}         g := 1;
       "Select the type of redundancy:
            1 - space
            2 - data
            3 - time"                                {1,2,3}       type := 3;
       "Enter the order of redundancy"               {3,5}         order := 3;
$
```

Fig. 6. Meta-interface of meta-program in Open PROMOL for generating fault-tolerant architectures.

```
$
"Select the type of redundancy:
             1 - space
             2 - data
             3 - time"                               {1,2,3}       type := 3;
$
@- the beginning of the meta-meta-body
       @sub[{
       $
        "Select a component:
             1 - dragonfly core
             2 - i8051 microcontroller
             3 - Free6502 core"                      {1,2,3\}      c := 1;
[c = 1] "Enter address bus width for program memory" {8,16\}       prgm := 16;
[c = 1] "Enter address bus width for data memory"    {4,8\}        dm := 8;
[c = 1] "Enter address bus width for i/o"            {4,8\}        io := 8;
[c = 3] "Generate Free6502 core version for:
             1 - synthesis
             2 - debugging"                          {1,2\}        g := 1;

        "Enter the order of redundancy"              {3,5\}        order := 3;
       $
             }]
       @- the rest part of the body at meta-meta-level which is omitted
 @- the end of the meta-meta-body (i.e., meta-meta-program)
```

Fig. 7. Meta-interface of two-level meta-program in Open PROMOL for generating fault-tolerant architectures.

The experiments with illustrative examples are not presented here because there is a motivating example (Fig. 5). Experiments for real tasks are presented in Table 1.

All experiments were checked using the Open PROMOL processor (Damaševičius, 2000) and theoretical prepositions were approved.

$M^{2*}$ is derived from Fig. 7 by moving the parameter *order* (see Fig. 7) to the meta-meta level. $M^{2**}$ is derived from Fig. 7 by exchange of parameters among the meta-meta-level and meta-level. All meta-specifications are equivalent (see Properties 1 and 3 in Section 7) because, when executed, they have produced the same programs. The total number of unique instances (programs) generated from each meta-meta-program was equal to 66 (see also (9)–(11)).

Table 1
Characteristics of meta-specifications of real tasks: a case study

| Meta-specification | Number of meta-meta-parameters | Number of meta-parameters | Cyclomatic index of meta-programs | Cyclomatic index of meta-meta-program | Explanation |
|---|---|---|---|---|---|
| $M^1$ | – | 7 | 66 | – | See Fig. 6 and (9) |
| $M^2$ | 1 | 6 | $22 + 22 + 22 = 66$ | 3 | See Fig. 7 and (10) |
| $M^{2*}$ | 2 | 5 | $6 * (8 + 1 + 2) = 66$ | 6 | At level 2: *type* and *order* |
| $M^{2**}$ | 6 | 1 | $22 * 3 = 66$ | 22 | Reverse structure of $M^2$ |

## 9. Summary, Discussion and Evaluation

We have analyzed some generalization aspects of heterogeneous meta-programs through transformation processes. First, we have presented the definition of basic terms, meta-programming-oriented transformation tasks, their properties and rules to support the transformation processes of the approach. The generalization is achieved through the introduction of an extra level of abstraction applied at the meta-specification's design stage according to prescribed requirements and anticipated model for change. We have used abstract rather than formal languages (meta- and target) as well as abstract models to specify transformation processes throughout the paper. Our intention was to formalize to some extent the abstract models and processes to achieve conciseness, strictness and unique interpretation. Thus, the discussed transformation processes should be conceived from the pure software engineering viewpoint rather than from the perspective of formal programming in computer science, where program correctness is based on the mathematically sound proofs.

We have demonstrated that meta-programming not only enables to develop generic programs to support generative reuse, i.e., building program generators. More specifically, we have shown that meta-programs *per se* can be generalized through the introduction of the extra meta-level within the internal structure of a given meta-program leading to the development of meta-meta-programs. Such a view of meta-programming raises the issue of transformations to transform a given target program into a meta-program and then, if it is needed, to transform the latter into a meta-meta-program. Note that the task related to the first type of transformation (i.e., program into meta-program) is beyond the scope of this paper.

The described transformations have a much wider context and consequences than purely technical aspects of generalization. As programming is also a social activity, the introduction of higher-levels (i.e., meta- and meta-meta-levels) changes the focus to meta-programming and the role of actors who perform this activity. The development of an executable meta-specification is the concern of a meta-designer or meta-programmer. He/she is also the owner of meta-specifications as an intellectual property. The use and adaptation of meta-specifications is the concern of a lower-level user (designer, programmer). The separation of actor roles leads to the new capabilities to manage the process in the development of products using meta-programming.

In general, reverse transformations describe the meta-(meta-)program development process. The introduced formalism enables to better understand the process and to make its implementation easier. As meta-programming is a hard task, by using the reverse transformation model, e.g., it is possible to introduce meta-parameters in a step-by-step manner at both levels, thus simplifying the process. In other words, the two-stage reverse transformation supports meta-program evolution. Note that meta-programs should be applied in such a context where variability (i.e., the scope of parameter space) is large and, at some extent, known in advance.

The next important issue is as follows: where the discussed executable meta-specifications, i.e., program generators, and their transformations can be applied? One answer follows – their most likely place is external component libraries and repositories for wide scale reuse, where a great variability of similar components can be found. This answer is based on two observations though there are many others: (1) the *library scaling* problem, which has been identified by Biggerstaff (1994); (2) the SPIRIT library concept (Martin, 2004), where generators are main constituents within the library though the used technology to implement generators is not revealed.

## 10. Conclusions

The reverse transformation process $R^2$, when a heterogeneous meta-program $M^1$ is transformed into the heterogeneous meta-meta-program $M^2$*within the same parameter space*, is *a semantics-preserving* transformation. First, $R^2$ can be seen as a meta-program *refactoring* process (in terms of software reverse engineering). Second, the construction of a meta-meta-program through the use of $R^2$ is a process for a *partial evaluation* of a meta-program (in terms of formal meta-programming). The forward transformation processes $F^2$ and $F^1$, when $M^2$ is first transformed into a set of meta-programs $M_i^1$, and then, each meta-program $M_i^1$ is transformed into sets of programs, are transformations preserving equivalent complexity in terms of Cyclomatic Index used as a complexity metric. The forward transformations $F^2$ and $F^1$ are also seen as meta-program and program generation processes, respectively. $R^2$ is the abstraction level lifting transformation. $F^2$ and $F^1$ are the abstraction level lowering transformations.

The presented formalism and identified properties can be treated as a background to build automatic tools for reverse transformations (e.g., to transform a meta-program into the meta-meta-program automatically).

**Acknowledgements.** Authors would like to thank anonymous reviewers for their comments that enabled us to revise the paper anew and to improve it substantially.

## References

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

Batory, D. (2004). Program comprehension in generative programming: a history of grand challenges. In: *Proceedings of 12th IEEE International Workshop on Program Comprehension (IWPC 2004)*, Bari, Italy, pp. 2–11.

Becker, M. (2001). Generic components: a symbiosis of paradigms. In: Butler, G., Jarzabek, S. (Eds.), *Second International Symposium on Generative and Component-Based Software Engineering (GCSE 2000)*, Erfurt, Germany, *LNCS*, Vol. 2177, pp. 100–113.

Benestad, A.Ch., Anda, B., Arisholm, E. (2009). Understanding software maintenance and evolution by analyzing individual changes: a literature review. *Journal of Software Maintenance and Evolution: Research and Practice*, 21, 349–378.

Biggerstaff, T. (1994). The library scaling problem and the limits of concrete component reuse. In: *Proceedings of the Third International Conference on Software Reuse: Advances in Software Reusability*, Rio de Janeiro, Brazil, pp. 102–109.

Boehm, B. (2010). The changing nature of software evolution. *IEEE Software*, 27(4), 26–28.

Buckley, J., Mens,T., Zenger, M., Rashid, A., Kniesel, G. (2003). Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5), 309–332.

Cheong, Y.C., Jarzabek, S. (1999). Frame-based method for customizing generic software architectures. In: *Proceedings of the Fifth Symposium on Software Reusability*, Los Angeles, CA, USA, pp. 103–112.

Cordy, J.R., Sarkar, M.S. (2004). Metaprogram implementation by second order source transformation. In: *Software Transformation Systems Workshop at Generative Programming and Component Engineering Conference (GPCE'04)*, Vancouver, Canada.

Damaševičius, R. (2000). *Open PROMOL – the Program Modification Language (Tutorial)*. Available at: `http://soften.ktu.lt/~stuik/group/promol/docs/`.

Damaševičius, R. (2005). *Transformational Design Processes Based on Higher Level Abstractions in Hardware and Embedded System Design*. PhD dissertation, Kaunas, Kaunas University of Technology.

Damaševičius, R., Štuikys, V. (2008). Taxonomy of the fundamental concepts of metaprogramming. *Information Technology and Control*, 37(2), 124–132

Damaševičius, R., Štuikys, V. (2010). Metrics for evaluation of metaprogram complexity. *ComSIS*, 7(4), 769–787.

Damaševičius, R., Genutis, M., Štuikys, V. (2004). Design of distributed generic embedded components. *Information Technology and Control*, 32(3), 61–65.

Ebraert, P., Classen, A., Heymans, P., D'Hondt, T. (2009). Feature diagrams for change-oriented programming. In: Nakamura, M., Reiff-Marganiec, S. (Eds.), *Feature Interactions in Software and Communication Systems X*, IOS Press, Amsterdam, pp. 107–122.

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

Fahmy, H., Holt, R.C., Cordy, J.R. (2001). Wins and losses of algebraic transformations of software architectures. In: *Proceedings of 16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, Coronado Island, San Diego, CA, USA, pp. 51–62.

Fischer, G., Scharff, E. (2000). Meta-design: design for designers. In: *Proceedings of the Conference on Designing Interactive Systems: Processes, Practices, Methods and Techniques*. ACM Press, New York, pp. 396–405.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, Reading.

Francis, M. (2004). Metagene, a C++ meta-program generation tool. In: *Proceedings of Multiparadigm Programming with Object-Oriented Languages (MPOOL'04)*, Oslo, Norway.

Iranzo, P.J. (2003). Thesis: Partial evaluation of lazy functional logic programs. *AI Communications,* 16(2), 121–123.

Janota, M., Kiniry, J. (2007). Reasoning about feature models in higher-order logic. *Procedings of 11th International Conference on Software Product Lines*, Kyoto, Japan, pp. 13–22.

Jarzabek, S., Pettersson, U. (2006). Research journey towards industrial application of reuse technique. In: *Proceedings of 28th International Conference on Software Engineering (ICSE'06)*, Shanghai, China, pp. 608–611.

Johnson, B.W. (1989). *Design and Analysis of Fault-Tolerant Digital Systems*. Addison-Wesley, Reading.

Jones, N.D., Gomard, C.K., Sestoft, P. (1993). *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York.

Karaila, M., Systa, T. (2007). Applying template meta-programming techniques for a domain-specific visual language. An industrial experience report. In: *Proceedings of 29th International Conference on Software Engineering (ICSE'07)*, Minneapolis, MN, USA, pp. 571–580.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J. (1997). Aspect-oriented programming. In: *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP'97)*, Jyväskylä, Finland. *LNCS*, Vol. 1241, Springer, Berlin, pp. 220–242.

Martin, G. (2004). IP reuse and integration in MPSoC: ighly configurable processors. In: *Tensilica Inc. Presentation at MPSoC'2004*, Hôtellerie du Couvent Royal, Saint-Maximin la Sainte Baume, France.

McCabe, T.J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4), 308–320.

Mens, T., Demeyer, S., Janssens, D. (2002). Formalising behaviour preserving program transformations. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (Eds.), *Graph Transformation, Proceedings of First International Conference*, Barcelona, Spain. *LNCS*, Vol. 2505, Springer, Berlin, pp. 286–301.

Montvilas, M. (2009). *Automated Development of Portals Considering Design for Change and Component Generation: Concept, Methodology and Implementation*. Summary of doctoral dissertation, Kaunas University of Technology.

Oram, A., Wilson, G. (2010). *Making Software: What Really Works, and Why We Believe It*. O'Reilly.

Ortiz, A. (2007). An introduction to metaprogramming. *Linux Journal*, (158).

Pataki, N., Sipos, Á., Porkoláb, Z. (2006). Measuring the complexity of aspect-oriented programs with multi-paradigm metric. In: *Proceedings of 10th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, Nantes, France, pp. 1–10.

Reis, G.D., Järvi, J. (2005). What is generic programming? In: *Proceedings of Workshop on Library-Centric Software Design (LCSD'05)*, Vol. 6(12), pp. 1–10.

Ross, K.D. (2006). Towards an automatic complexity analysis for generic programs. In: *Proceedings of the 2006 ACM SIGPLAN Workshop on Generic Programming*, Portland, Oregon, USA.

Schordan, M., Quinlan, D.J. (2005). Specifying transformation sequences as computation on program fragments with an abstract attribute grammar. In: *Proceedings of 5th IEEE International Workshop on Source Code Analysis and Manipulation*, Budapest, Hungary, pp. 97–106.

Sheard, T. (2001). Accomplishments and research challenges in meta-programming. In: *Proceedings of 2nd International Workshop on Semantics, Application, and Implementation of Program Generation (SAIG'2001)*, Florence, Italy, *LNCS*, Vol. 2196, Springer, Berlin, pp. 2–44.

Sobiesiak, R., Diao, Y. (2010). *Quantifying Software usability Through Complexity Analysis*. IBM technical report. Available: `http://www-01.ibm.com/software/ucd/Resources/Complexity AnalysisDGarticle.pdf`.

Štuikys, V., Damaševičius, R. (2003). Metaprogramming techniques for designing embedded components for ambient intelligence. In: Basten, T., Geilen, M., de Groot, H. (Eds.), *Ambient Intelligence: Impact on Embedded System Design*, Kluwer, Dordrecht, pp. 229–250.

Štuikys V., Damaševičius, R., Ziberkas, G. (2002). Open PROMOL: an experimental language for target program modification. In: Mignotte, A., Villar, E., Horobin, L. (Eds.), *System on Chip Design Languages – Extended Papers: Best of FDL'01 and HDLCON'01*, Kluwer, Dordrecht, pp. 235–246.

Štuikys, V., Montvilas, M., Damaševičius, R. (2009). Development of web component generators using one-stage metaprogramming. *Information Technology and Control*, 38(2), 108–118.

Taha, W. (1999). A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. *ACM SIGPLAN Notices*, 34(11), 34–43.

Trujillo, S., Batory, D.S., Díaz, O. (2007a). Feature-oriented model driven development: a case study for portlets. In: *Proceedings of 29th International Conference on Software Engineering (ICSE 2007)*, Minneapolis, MN, USA, pp. 44–53.

Trujillo, S., Azanza, M., Díaz, O. (2007b). Generative metaprogramming. In: *Proceedings of 6th International Conference on Generative Programming and Component Engineering (GPCE 2007)*, Salzburg, Austria, pp. 105–114.

Veldhuizen, T.L. (2006). Tradeoffs in metaprogramming. In: *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. Charleston, SC, USA, pp. 150–159.

Vidacs, L. (2009). *Software Maintenance Methods for Preprocessed Languages*. Summary of the PhD dissertation, Institute of Informatics, University of Szeged.

Visser, E. (2001). A survey of strategies in program transformation systems. In: Gramlich, B., Lucas, S. (Eds.), *Electronic Notes in Theoretical Computer Science*, 57, pp. 144–162.

Welinder, M. (1996). *Partial Evaluation and Correctness.* PhD thesis, Department of Computer Science, University of Copenhagen, Denmark.

Westfechtel, B., Conradi, R. (2009). Multi-variant modeling concepts, issues, and challenges. In: *1st International Workshop on Model-Driven Product Line Engineering (MDPLE'2009) at European Conference on Model-Driven Architecture (ECMDA)*, Twente, The Netherlands, pp. 57–67.

Winter, V.L. (2004). Program transformation: what, how and why. In: Wah, B.W. (Ed.), *Wiley Encyclopedia of Computer Science and Engineering*, Wiley, New York.

Yang, C. (2009). Analyzing the influences of aspect weaving on software system behavior. In: *Proceedings of International Symposium on Web Information Systems and Applications (WISA'09)*, Nanchang, China, pp. 234–237.

**V. Štuikys** is a professor at Software Engineering Department of Kaunas University of Technology (KTU), Kaunas, Lithuania. He received the PhD and doctor habilitatis titles from KTU in 1970 and 2002, respectively. He is a lecturer and a researcher as well as a leader of the Design Process Automation Group. His research interests include software and hardware design methodologies, software reuse, component-based programming, meta-programming and program generation, CAD systems and soft IP design. He has published more than 100 papers in the area. He is an author of several books and a monograph. He is a member of ACM and IEEE.

**R. Damaševičius** is a professor at Software Engineering Department, KTU. He received his PhD degree in informatics engineering from KTU (2005). Currently he teaches several computer science, programming and software engineering courses. He is also the member of Design Process Automation Group at Software Engineering Department. His research interests include program transformation and meta-programming, design automation and software generation, as well as domain analysis methods.

## Heterogeninių meta-programų ekvivalenčios transformacijos

Vytautas ŠTUIKYS, Robertas DAMAŠEVIČIUS

Straipsnyje pateikiamas transformacijomis grindžiamas heterogeninių meta-programų apibendrinimas, kai meta-programos vienpakopė struktūra keičiama į dvipakopę struktūrą, esant tai pačiai meta-parametrų aibei. Apibrėžiami pagrindiniai terminai, formalizuojami transformavimo uždaviniai, specifikuojamos transformacijų savybės ir taisyklės. Programų inžinerijos požiūriu analizuojami tokie procesai: (1) apgrąžos transformacija, kai korektiška vienos pakopos meta-programa $M^1$ yra transformuojama į ekvivalenčią dviejų pakopų meta-meta-programą $M^2$; (2) dvipakopė tiesioginė transformacija, kai iš pradžių $M^2$ yra transformuojama į meta-programų poaibį, o po to kiekvienas poaibis yra transformuojamas į programų aibę. Pagrindiniai rezultatai tokie: (1) heterogeninių meta-programų transformavimo proceso formalizavimas; (2) ekvivalenčių transformacijų iš $M^1$ į $M^2$, ir atvirkščiai, aprobavimas; (3) meta-specifikacijų sudėtingumo įvertinimas. Rezultatai pagrindžiami pavyzdžiais, teoriniais teiginiais bei eksperimentais.