ISSN 1392–124X (print), ISSN 2335–884X (online) INFORMATION TECHNOLOGY AND CONTROL, 2014, T. 43, Nr. 1

# Refactoring of Heterogeneous Meta-Program into *k*-stage Meta-Program

**Vytautas Štuikys, Kristina Bespalova, Renata Burbaitė**

*Software Engineering Department, Kaunas University of Technology, Lithuania*
*e-mail: vytautas.stuikys@ktu.lt, kristina.bespalova@stud.ktu.lt, renata.burbaite@stud.ktu.lt*

crossref http://dx.doi.org/10.5755/j01.itc.43.1.3715

**Abstract**. The paper presents: (1) a graph-based theoretical background to refactoring a correct heterogeneous meta-program into its *k*-stage representation; (2) the refactoring method; (3) refactoring experiments with tasks taken from different domains, including real world tasks, such as meta-programs to teach Computer Science (CS) topics using educational robots. Refactoring meta-programs by staging enables to flexibly adapt them to the different context of use. To do that (semi-)automatically, we use the contextual information as a priority relation (e.g. highest, lowest, etc.) introduced within the meta-program specification. We implement the refactoring method using the so-called activating/de-activating label (index) to change the role of meta-language constructs at different stages. The contribution of the paper is: (1) applying the known (in programming) staging concept to heterogeneous meta-programming; (2) a theoretical background, properties and the method to solve tasks of this kind of refactoring.

**Keywords**: refactoring; meta-program; meta-parameter; meta-programming; multi-stage heterogeneous meta-program.

## 1. Introduction

Refactoring is the process that takes an existing program and transforms it into an improved new version. Refactoring changes the program's structure only, without affecting its external behaviour. The improvements typically eliminate redundancy, bad smell of code [25], improve maintainability and may improve performance and reduce space [10, 22].

In this paper, we consider refactoring of the heterogeneous meta-program into its representation which we call *k*-stage meta-program. In general, meta-programming is such a programming paradigm that deals with program transformations [13], or writing programs that generate other programs [3, 15, 16, 23]. Powerful meta-programming is essential for approaches to automating the software development. Most domain-specific languages and many other software automation tasks are implemented as program generators. At the core of meta-programming is the explicit representation and specification of domain variability. In the case of heterogeneous meta-programming, we express variability aspects at a higher-level of abstraction through parameters using a meta-language, while the base domain functionality we express using a target (domain) language. Thus, heterogeneous meta-program is a domain program generator [44].

Similarly to program refactoring, by refactoring a meta-program into its *k*-stage representation, we are also seeking for improvements; however, they are of the quite different sort. For example, *k*-stage meta-program extends generative reuse lifting it to the meta-level (qualitative improvement). By constructing the *k*-stage meta-program, we construct a *meta*-generator that generates the lower-level meta-programs. The program generator and meta-generator (also known as meta-meta-program) are a value in own right, though they cannot be constructed and used in any case. Typically the degree of base (functional) variability of a domain and its context variability predefines the motivated use of meta-programming approaches. Usually, at the specification level, researchers and practitioners represent variability using feature models [7] where variability is modelled by variation points and variants. We consider a few domains with such features, where meta-programming-based generators have been approved as a relevant technology. Examples are: e-learning, e-commerce and components of embedded systems [6, 44].

In the e-learning domain, for example, we use multi-stage generative learning objects (Computer Science (CS) teaching content that is represented as meta-programs [41]). In this case, the *k*-stage meta-programs enable to flexibly (automatically) adapt the generated content to different context of use (e.g. the teacher's context, such as teaching objectives and teaching model, the students' context (abilities,

previous knowledge, etc.), the e-learning environment context (use PC only, educational robots [30], etc.)).

In contrast to program refactoring, the structure of *k*-stage meta-program after refactoring becomes more complex as compared to its initial form because we need additional facilities to manage staging (designer's viewpoint). However, from the user perspective, the *k*-stage meta-program may look much simpler because he/she actually works with the user-friendly meta-interface that, at each stage, has the reduced number of meta-parameters while the implementation hides staging fully. In fact, we can interpret the *k*-stage meta-program as a tool to manage complexity by compressing information (programs, meta-programs) into the single high-level specification, and then, sharing the separable parts of the specification through the generation process in the different context of use.

The aim of the paper is to propose a theoretically-grounded method to re-factor a meta-program into its *k*-stage format and approve the method experimentally (e.g. equivalence of such transformations) before developing the automatic refactoring tool. Such a tool serves for changeability (adaptation) and maintenance of meta-programs we use in different domains (e.g. e-learning in CS [40], e-commerce system design [21]).

The idea of *multi*-stage programming is not new. We have borrowed it from [1, 47, 48]. However, we apply it in the new context, the heterogeneous meta-programming domain, pursuing other aims. We focus on generative reuse through the *pre-programmed anticipated variability* (at design time) and possibility to flexibly adopting the highly reusable meta-programs to the different context of use by staging (at use time) while other authors focus on performance issues.

We believe that, by introducing *k*-stage meta-programs, we can not only gain practical benefits (see case study). We are also able to achieve methodological aims such as better understanding of heterogeneous meta-programs, improving their design and maintenance methods and tools.

The paper's main contribution is a theoretical background of meta-program refactoring that includes: formal definitions of basic terms, rules, properties of models to perform transformation/refactoring processes. We have also obtained the upper bound of the number of stages the given meta-program can be re-factored into the *k*-stage representation. On this background, we have developed the method to refactoring the given correct meta-program into *k*-stage meta-program, thus constructing meta-generators for particular domains.

The paper's structure is as follows. Section 2 analyses the related work. Section 3 presents the informal description of the approach with running examples. Section 4 describes the theoretical background. Section 5 provides the refactoring method along with experimental results. Section 6 delivers a discussion and evaluates the approach. Section 7 concludes the main results.

## 2. Related Work

Manipulation of the program source code is one of the main aspects of any software development process. As the software content in systems steadily increases [4] and the complexity of software grows continuously, the manual manipulation becomes ever more infeasible. On this account, Batory [9] evaluates software trends as "the *alarming complexity* of software and the *alarming rate* at which software complexity is increasing". It is the reason why researches and practitioner make a strong focus on higher-level modelling (see e.g. MDA [24], Product Line Engineering (PLE) [19, 20]) approaches and higher-level programming now.

Meta-programming is just the case. It is a higher-level programming paradigm which deals of how manipulating programs as data. The result of the manipulation is the lower-level program. There are many different views to understand, to study or to deal with this approach (reader can learn more from [44]). For example, according to Veldhuizen [37], meta-programming can be seen as a program generalization and generation technique. The meta-programming taxonomies [12, 35] provide a systemized knowledge on the topic. Sheard [35] reviews also research challenges, describes formal meta-programming systems and their role in program generation. Meta-programming also contributes to complexity management. The more we use program generators, the less code we need to write manually. Hence, the more complex software systems can be developed. Furthermore, their quality is better, and time-to-market is shorter.

In large, meta-programming is the domain to research model transformations too. Mens *et al.* [33] state in this regard that the term "model transformation" encompasses the term "program transformation" since a model can range from the abstract analysis models, over more concrete design models, to very concrete models of the source code. Visser [13] presents a taxonomy that considers two major groups of transformations: translation and rephrasing. Winter [38] identifies seven major bi-directional goals of program transformation: clarity, efficiency, computability, simplicity, functionality, translation, and computation. Cordy and Sarkar [18] demonstrate that meta-programs can be derived from higher-level specifications using second order source transformations. Trujillo *et al.* [32] 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. Transformation of meta-programs leads to the problem of measuring their semantic equivalence.

Formal and semi-formal description of meta-programs, meta-programming and related higher-level programming methodologies and transformations for implementing higher-level programs has been intensively studied by many researchers. Though meta-programming was known and used for a long time in formal logic programming [12], now, however, the scope of the application of the meta-programming techniques is much wider. These include the domains such as programming language implementation, parser and compiler generation [29], application and software generators [8], product lines, program transformations [2], generative reuse, XML-based web applications and web component deployment [46]. Many, if not all of the presented cases, can be summarized as multi-stage programming, i.e. the development of programs in several different stages.

Taha was the first to provide a formal description for a multi-stage programming language [48]. Staging is a program transformation that involves reorganizing the program execution into stages. He treats the use of the formal language MetaML to develop meta-programs as multi-stage programming. The concept relates to the fundamental principle of information hiding through the introduction of a set of abstraction levels (stages) aiming at gaining a great deal of flexibility in managing the program construction process. In this approach (similarly to partial evaluation [26]), the program's performance optimization is a main focus.

However, in the case of heterogeneous meta-programming, we can use any programming language satisfying a set of minimal requirements in the role of a meta-language (has abstractions for output, looping, etc.) [45]. Application domain and designer's flavour are the most decisive attributes for selecting the languages.

Refactoring is a program transformation aiming at improving the structure of a program without changing its behaviour [10, 22]. Reader can learn more on this topic from the comprehensive survey [34]. The paper [31] considers the refactoring problem at the level of feature models. As meta-programs implement feature models, both topics are equally important, but here we focus on the first topic.

Though there is a great deal effort on program refactoring research (see e.g. [5, 34]), to our best knowledge, meta-program refactoring is restricted either by homogeneous meta-programming (such as logic meta-programming [36]), or aspect-oriented programming [25].

Though our approach has some conceptual similarities with the ones discussed above, it has also essential differences as follows. We focus on: 1) using general-purpose languages as a meta-language for heterogeneous meta-programming; 2) staging is concerned with the generation stage (but not with the execution stage as it is the case in the Taha's approach); 3) automatic adaptation (reuse) through generation, but not on program performance. Some partial results on meta-program refactoring are also given in [39, 43].

## 3. Basic idea and running examples

Meta-programming and the development of heterogeneous meta-programs can be dealt with and understood using two general engineering approaches: forward engineering and reverse engineering [42]. The latter approach enables to re-factor a given heterogeneous meta-program and represent it as a *k*-stage meta-specification aiming at developing meta-meta-programs or meta-generators, or pursuing other goals. As the complexity of systems and their components is steadily increasing, meta-programs and *k*-stage meta-programs can be seen as tools for managing complexity and changeability aiming to adapt generated programs to the different context of use.

The use of models constitutes a background for transformations. To explain the transformations, we first introduce motivating examples, and then the structural models of the 1-stage and *k*-stage meta-programs. Say, we aim at developing a simple meta-program for generating a set of the homogeneous Boolean equations (they are instances of a target program here). We represent two variants of the set as follows:

Variant 1: Y = X1 AND X2; Y = X1 OR X2 OR X3 OR X4; Y = X1 AND X2 AND X3;

Variant 2: Y = NOT (X); Y = X1 AND X2; Y = X1 OR X2 OR X3 OR X4; Y = X1 AND X2 AND X3.

The meta-program has to specify the *variable aspects* (i.e. variability) of the domain of Boolean equations using a meta-language while the base aspects (i.e. commonality) have to be expressed using a target language (in our case, the assignment statement: the left side variable, symbol "=", and the expression to the right). We code variability by (meta-) parameters. For Variant 1 (we explain Variant 2 later), there are two parameters: P1 and P2. The first (meaning *operation*) has two values (AND, OR). The second (meaning the *number of arguments*) has 3 values (2, 3, and 4). In Fig. 1(a), we present the 1–stage meta-program's model. In Fig. 1(b), we give its full implementation in PHP (used as meta-language). In Fig. 1(c), we outline the internet-based representation of the meta-program as the user sees it. The meta-program, when interpreted by the PHP processor, can generate any instance on demand from the space of possible variants (in our case 2*3=6). The generated instance (when P1=AND and P2=3) is given at the bottom (see Fig. 1).

One can predict that the use of the *k*-stage meta-program is beneficial when the number of parameters is large enough. We can enlarge, for example, the variability space of Variant 1 by introducing *changes* for the function name (Y, Z) and for the assignment operator ("=", ":="). However, even without the

| a) | b) | c) |
|---|---|---|
| Meta-interface<br>of Meta-program | `<?php`<br>`// here is meta-interface`<br>`$P1 = "AND";`<br>`$P2 = 3;` | Select a function: AND (OR, AND)<br>Enter the number of arguments: 3 (2, 3, 4)<br>Submit value |
| Meta-body<br>of Meta-program | `// here is meta-body`<br>`echo "Y = X"."1";`<br>`for($i = 2; $i <= $P2; $i++)`<br>`echo " $P1 X".$i;`<br>`?>` | Meta-body as a Black Box (invisible part) |
| | The generated instance: | Y = X1 AND X2 AND X3 |

**Figure 1.** Meta-program model (a), its full implementation in PHP for Variant 1 (b) and user's vision (c)

| a) | b) | c) |
|---|---|---|
| `<?php`<br>`// here is meta-interface of stage 2`<br>`$P2 = 3;` | Enter the number of arguments: 3 (2, 3, 4)<br>Submit value | Meta-interface of *k*-stage meta-program<br>Meta-body of *k*-stage meta-program<br>. . .<br>. . .<br>Meta-interface of 2-stage meta-program<br>Meta-body of 2-stage meta-program<br>Meta-interface of 1-stage meta-program<br>Meta-body of 1-stage meta-program |
| `//here is the meta-body of stage 2`<br>`echo "<?\n";`<br>`echo "\$P1 = \"AND\";\n";`<br>`echo "echo \"Y= X\".\"1\";\n";`<br>`echo "for(\$i=2;\$i<=$P2; \$i++)\n";`<br>`echo "echo \" \$P1 X\".\$i;\n";`<br>`echo "?>\n";`<br>`?>` | Meta-body of 2-stage as a Black Box<br>Select a function: AND (OR, AND)<br>Submit value<br>Meta-body of 1-stage as a Black Box | |

**Figure 2.** 2-stage meta-program for Variant 1 (a), user's vision of it (b), model of *k*-stage meta-program (c)

enlargement, the given meta-program can be rewritten as the 2-stage meta-program (see Fig. 2(a)). Fig. 2(b) represents the user's view of the 2-stage meta-program. The role of the symbol "\" is to manage interpretation of the target program at stage 2 (see DEFINITIONS 4-7, for details). Fig. 2(c) represents the *k*-stage meta-program model containing the *multi-level meta-interface* and *multi-level meta-body*.

Though the basic idea is clear from the running examples, the transformation is not a trivial task. It requires as much as deep insights to study as follows.

## 4. Theoretical background of meta-program transformations

### 4.1. Basic definitions

Meta-program is a higher-level executable specification $M$ (aka program generator), which is coded using two languages (meta-language $L_M$ and target language $L_T$), to specify and generate a set of programs in $L_T$. The $L_M$ processor (compiler) is the transformation tool to derive or generate programs in $L_T$ from the meta-program. To understand the syntax (structure) of any meta-program (1-stage, *k*-stage), it is enough to read and study running examples given in the paper. As running examples are supplemented by the result of executing $M$, one is also able to understand semantics (behaviour) of the meta-program. However, the deep understanding of meta-program refactoring necessitates more precise models. We introduce them through the following definitions and notions.

We denote 1-, 2-, and *k*-stage meta-programs as $M^1$ (also $M$), $M^2,\ldots, M^k$ respectively. Meta-program is a structure that consists of the meta-interface and the meta-body (see Fig. 1(a)). Also the meta-program is a target program generator. The *k*-stage meta-program is a structure that consists of the *multi-level meta-interface* and the *multi-level meta-body* (see Fig. 2(c)). The meta-interface (single or the multi-level) specifies (meta-)parameters and their values. The 2-stage meta-program is a meta-meta-program or meta-generator. The *k*-stage meta-program is the meta-meta-generator.

As, in fact, parameters play a decisive role in refactoring, we need to introduce the following formal definitions.

DEFINITION 1. In terms of the set-based notion, meta-interface model $\mu(M_I)$ of $M$ is the $n$-dimensional non-empty (meta-)parameter space $\mathcal{P}$:

$$\mu(M_I) = \mathcal{P}, \quad (1)$$

where $\mathcal{P} = \{P; V\}$, $P$ – the full set of $n$ meta-parameter names, i.e. $n = |P|$, $V$ – the ordered set of *all* meta-parameter values.

As each meta-parameter $P_i (P_i \in P)$ has its own set of values $\{v_{i_1}, v_{i_2}, ..., v_{i_q}\} \subset V$ , we can write:

$$P_i := V_i = \{v_{i_1}, v_{i_2}, ..., v_{i_q}\} \in V, \quad (2)$$

where $i_q$ – the number of values of meta-parameter $P_i$ . The symbol ":=" means 'is defined'.

DEFINITION 2. Two meta-parameters $P_i$ and $P_j$ ( $P_i, P_j \subseteq P$ $(i \neq j)$ ) are said to be *independent upon the choice of their values*, if any pair of values $\{v_{i_d}, v_{j_t}\}$ $(v_{i_d} \in P_i, v_{j_t} \in P_j$ , where $d \in [1, i_q]$ and $t \in [1, j_m]$ ) can be selected to correctly evaluate the specification $M$, when it is executed. Otherwise, the meta-parameters are *dependant* upon the choice of their values.

Sometimes dependent meta-parameters are treated as interacting (especially in terms of aspects or features [44]).

Note that this definition defines the *one-to-one* dependency in making a choice of meta-parameter values. The whole space $\mathcal{P}$ is used to constructing the meta-parameter dependency graph $G(P,U)$ as follows. The set of nodes $P$ corresponds to meta-parameters. The set of edges $U$ is defined as follows: for all $i$ and $j$ $u_{ij} = 1$ (meaning the edge exists) **iff** two parameters $P_i$ and $P_j$ are dependable according to DEFINITION 2, otherwise $u_{ij} = 0$ (meaning the edge does not exist) ( $P_i, P_j \in P$ , $u_{ij} = (P_i, P_j) \in U$ ).

DEFINITION 3. In terms of the graph-based notion, the graph $G(P,U)$ is the meta-interface model $\mu^*(M_I)$ defined by Eq. (3).

$$\mu^*(M_I) = G(P,U) . \quad (3)$$

Firstly, $\mu^*(M_I)$ is the derivative model that has been derived from Eq. (1) (it follows from DEFINITION 1 and DEFINITION 2). Secondly, the model $\mu^*(M_I)$ is more precise (as compared to (1)) because it specifies the parameter dependency explicitly. As it will be clear later, this attribute is key to identify some useful properties in devising formal transformation rules. The left part of Fig. 3(a) and (b) explains the model $\mu^*(M_I)$ for our running examples (Variant 1 and Variant 2 respectively).

So far, we have defined the structural models of the meta-programs $M$ and $M^k$ . The behavioural (aka functional) model of $M^k$ is to be understood as follows. When $M^k$ is executed, the $L_M$ processor produces either a set of ($k$-1)-stage meta-programs, or a single ($k$-1)-stage meta-program, each dependent upon the pre-specified meta-parameter values.

To specify the functional model in designing meta-meta-programs, we need to introduce some technological terms such as *de-activating label*, *de-activating index*, *active/passive meta-construct*.

DEFINITION 4. Meta-construct (i.e. meta-parameter or meta-function of a meta-language within the meta-body) is *active* if it performs the pre-scribed action at the current stage defined by the meta-language. Simply, the active meta-construct has no de-activating label (see Fig. 1(b)).

Note that modern high-level languages (such as Java, C++, PHP, etc.) have the de-activating labels (denoted as "\") to control and change the role of language constructs during their compilation.

DEFINITION 5. Meta-construct is *passive* if it contains the de-activating label (labels) written before the meta-construct (see Fig. 2 (a)).

Note that if a meta-construct is passive at the current stage, the $L_M$ processor does not interpret the construct treating it as a target language text.

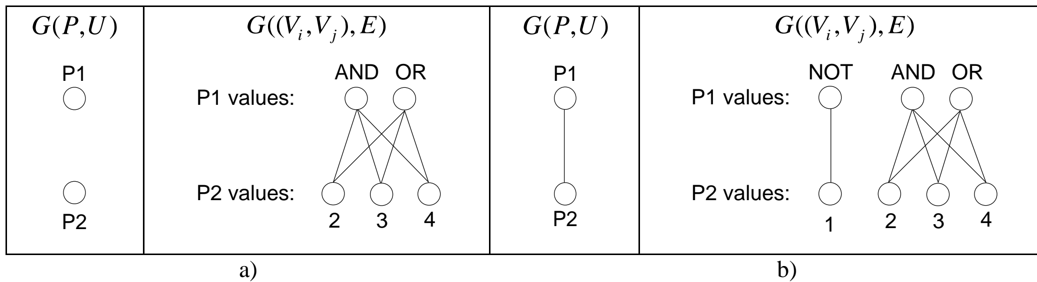


**Figure 3.** The parameter dependency and value interaction graphs for Variant 1 (a) and Variant 2 (b)

DEFINITION 6. *De-activating index* is the adequate number of de-activating labels written before a meta-construct. The value of the index depends on the meta-construct's stage and meta-language used (see RULE 5 in sub-section 4.4).

DEFINITION 7. *De-activating process* is the multi-stage process (in terms of *k*-stage processing) to reducing the de-activating index by 1, or changing the state of a meta-function from the passive state to the active state.

The $L_M$ processor performs the de-activating process reducing the de-activating index by 1 at the given stage. Note also that the de-activating process does not affect semantics (functionality or behaviour) of a meta-function. The process affects the meta-function's state only.

DEFINITION 8. Refactoring is the transformation process that alters the structure of a program without altering its observable behaviour [22]. We use this definition also for the meta-program refactoring.

DEFINITION 9. Reverse transformation of $M$ into $M^k$ ($M^k \xleftarrow{\mathcal{T}} M$) is the process $\mathcal{T}$ of refactoring the meta-program $M$ so that the model $\mu^*(M_I)$ (see DEFINITION 3) is transformed into the multi-stage meta-interface and the meta-body of $M$ is transformed into the multi-stage meta-body using the prescribed transformation rules (see sub-section 4.4).

DEFINITION 10. Forward transformation is the generating processes $\mathcal{G}$ defined as two cases as follows.

Case 1: $M^k \xrightarrow{\mathcal{G}} M^{k-1} \xrightarrow{\mathcal{G}} M^{k-2}$ …, where $M^{k-1}$ is either a single (*k*-1)-stage meta-program (if a single choice of meta-parameter values has been taken at stage *k*), or a set (subset) of the meta-programs (if multiple choices have been taken at stage *k*);

Case 2: $M^1 \xrightarrow{\mathcal{G}} R$, where *R* is either an instance of the target program (if a singe choice of meta-parameters values has been taken), or a set (subset) of the target programs (if multiple choices have been taken).

It is clear that Case 1 defines the process of generating meta-generators while Case 2 defines the process of program generators.

### 4.2. Formulation of transformation tasks

Now, having the formal definitions of basic terms, we are able to formulate tasks we consider in this paper.

Given: (*i*) meta-program model (see Fig. 1(a); Eq. (3) represents the key part of the model), (*ii*) specification $M$ that implements the model (*i*) and (*iii*) *k*-stage meta-program model (see Fig. 2(c)).

**Task 1** is to perform the reverse transformation to refactoring $M$ into the specification $M^k$ according to DEFINITION 9.

**Task 2** is to perform the forward transformations to generating either the lower-level meta-program(s), or target program(s) according to DEFINITION 10.

### 4.3. Graph-based background to specify refactoring

First we identify the conditions and properties to specify the meta-parameter (further parameter) dependency graph $G(P,U)$. Let be given the bipartite graph $G((V_i,V_j),E)$ (the parameter values interaction graph) defined for two parameters $P_i$ and $P_j$ $(i \neq j)$ as follows: edges $e_{dt} = (v_{i_d}, v_{j_t})$ $(v_{i_d} \in V_i, v_{j_t} \in V_j)$ specify the value interaction of the type $v_{i_d}$ requires $v_{j_t}$ (meaning $e_{dt} = 1$), or the interaction of the type $v_{i_d}$ excludes $v_{j_t}$ (meaning $e_{dt} = 0$).

We illustrate the bipartite graphs for our running examples (see the right parts of Fig. 3(a) and (b)). As the graphs specify the interaction (dependency) among parameter values, we call them value graphs. The bipartite value graphs serve to specifying some key properties of the parameter dependency graph $G(P,U)$ as follows.

PROPERTY 1. It is expressed by Eqs. (4) and (5) as follows. The parameter dependency graph $G(P,U)$ is the null graph (see Fig. 3(a)) **iff** for each pair of parameters $P_i, P_j \in P$ (*i*≠*j*) their value graphs are complete bipartite graphs:

$$\forall_b (G_b((V_i,V_j),E) \text{ is complete}) = \textbf{true}, \qquad (4)$$

where ( $b \in [1,|B|]$; $|B| = C_n^2$ ); $B$ – the number of different parameter pairs.

The parameter dependency graph $G(P,U)$ is disconnected (i.e. containing a set of connected components) **iff** the following property holds:

$$\exists_b (G_b((V_i,V_j),E) \text{ is non-complete}) = \textbf{true}. \qquad (5)$$

The parameter dependency graph can be expressed as:

$$G(P,U) = \bigcup_{i=1}^{g} G_i, \ (G_i \cap G_j = \varnothing;\ G_i, G_j \subseteq G\ ) \qquad (6)$$

$(i \neq j)$, $g$ is the number of connected components including isolated nodes ($g > 1$).

Fig. 4 presents some typical examples of $G(P,U)$: (a) – all parameters are independent as it is stated by Variant 1, see Section 3 and Fig. 1(b); (b) on left – the case indicating groups of dependent parameters as Variant 2 (e.g. such a case can be derived from Variant 1 by adding a new value (NOT) for the

function name, see Section 3), on right – there is a variant with 3 connected components; (c) – theoretically possible variant when the number of parameters is equal to 6.

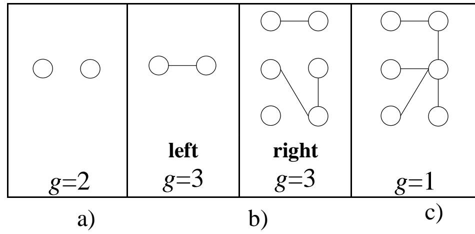


**Figure 4**. Graphs $G(P,U)$: (a) Variant 1, (b) left - Variant 2, right – other possible variant, (c) theoretically possible variant when $n$=6

Connected components $G_i, G_j \subseteq G$ define constraints (requirements) to specify stages of refactoring as follows.

PROPERTY 2. The upper bound of the eligible number of stages $k_{\max}$ to perform refactoring of a given (correct) meta-program into its $k$-stage format is defined by inequality (7):

$$k_{\max} \le g\,. \tag{7}$$

Now we are able to formulate the condition for solving Task 1.

**Statement.** Transformations $M^k \xleftarrow{\mathcal{T}} M$ $(1 < k \le k_{\max})$ exist **iff** the dependency graph $G(P,U)$ of $M$ is disconnected, i.e. defined by Eq. (6).

The proof is based on PROPERTY 1 and PROPERTY 2.

**Inference.** In fact, Eqs. (8) and (9) give the number of possible transformations $\mathcal{T}$ when $k$=2 and $k$=3, respectively:

$$|\mathcal{T}| = 2^g - 2, \tag{8}$$

$$|\mathcal{T}| = 3^g - \sum_{i=1}^{g} 3 * 2^{(i-1)}. \tag{9}$$

Eqs. (8) and (9) can be easily checked on examples taking into account PROPERTY 5 (see sub-section 4.4).

### 4.4. Properties and rules to support refactoring

Properties of the parameter dependency graph $G(P,U)$ are essential to form transformation rules. This graph, at the meta-program $M$ coding level, can be constructed due to the following property.

PROPERTY 3. The independent parameters are expressed through the *assignment statements*, while dependent parameters are expressed through the *conditional assignment statements* written within the meta-interface of $M$.

PROPERTY 4. The connected components $G_i \subseteq G(P,U)$ $(i = [1, g])$ (see Eq. (6)) represent groups of independent parameters.

PROPERTY 5. Any combination of parameter groups (i.e. $G_i$ ) can be lifted from stage 1 to any stage $k$ and evaluated there when $M$ is re-factored into $M^k$ .

All these properties are based on the background given in sub-section 4.3, where the dependency relation (i.e. edges of $G(P,U)$) has been constructing using the parameter value interaction (i.e. using the graph $G((V_i, V_j), E)$). However, analyzing the real world meta-programs, we have obtained yet another kind of the parameter interaction, which we call the *priority-based* parameter dependency. We explain that below.

Take, for example, the *Line Follower* task [17] that was implemented as a meta-program to describe different aspects of using educational NXT Robots to teach CS topics. We express these aspects through the following parameters (their values are in square brackets) [40]:

1. Teaching method (T): [project-based, problem-based];
2. Algorithm (A) type to follow the line by the Robot: [A1, A2, A3, A4] (e.g. A1 means the line following by zigzags using the only one light sensor);
3. 1st Light sensor (L1): [S1, S2, S3, S4] (S*i* means inputs of the NXT Intelligent Brick [27]);
4. 2nd Light sensor (L2): [S1&S2, S1&S3, S1&S4, S2&S3, S2&S4, S3&S4];
5. Selected Motor (S): [A&B, B&C, A&C] (A, B, C: outputs of the NXT Intelligent Brick, or names of motors);
6. Velocity (V) of motors in % calculated of maximum value: [10, 20, 30].

It seems that there is no other way to define the priority-based relation of parameters as taking into account the application context (i.e. semantics of the task). It can be introduced, for example, through categorizing parameters according to their priority levels. How many priority levels are needed? It depends upon the domain task and the intension of meta-designer whose responsibility is, at the design phase, to develop a meta-program and to anticipate the possible variants for the context adaptation to be provided by the user through refactoring, at the use phase. In general case, there might be the following priority levels: highest (HL), intermediate (IL), lowest (LL), and null priority. It is convenient to model the priority levels by colouring the nodes of the connected components using 4 colours as follows: black (for HL), dark (for IL), moderate dark (for LL) and white (for null priority). Fig. 5(b) illustrates colouring of the priority-based graph $G^*(P,U)$ for the *Line Follower* task.

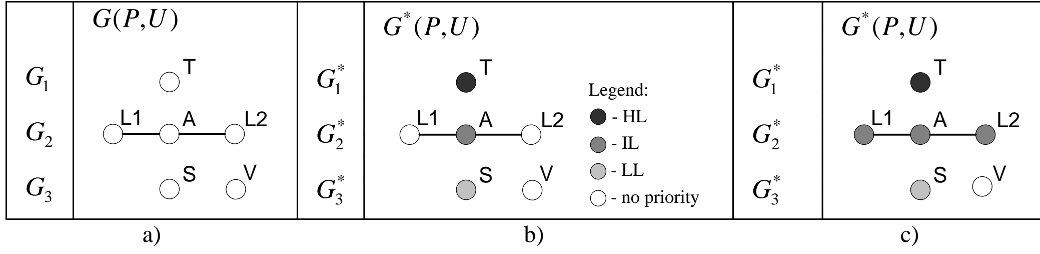


**Figure 5.** Graph $G(P,U)$: (a) – no priority (1-stage case), (b) – with priority nodes, (c) – with priority connected components

PROPERTY 6. If a connected component $G_i^*(G_i^* \subset G^*(P,U))$ has a coloured node (see Fig. 5(b)), then the remaining nodes of the connected component $G_i^*$ have the same colour (colour with a highest priority, see Fig. 5(c)) because, according to PROPERTY 5, they will appear at the same stage.

Now we are able to connect the parameter priority with stages because the *number* of a stage is actually its *priority*. For example, stage *k* is the highest and stage 1 is the *lowest*.

Thus, the following rule is valid.

RULE 1. The *HL-coloured* connected component $G_i^*$ (if any) has to be allocated to stage *k*, and the *LL- coloured* connected components $G_i^*$ (if any) have to be assigned to stage 1. The *IL-coloured* connected component(s) $G_j^*$ (if any) should be assigned in the stages between stages 1 and *k*. If there are no such stages, the graphs $G_j^*$ are moved to stage 1.

Returning to our example and applying RULE 1, we have the following feasible assignments of connected components to stages.

**Case 1** (when *k*=3 due to Eq. (7)): T - at stage 3; (S, V) – at stage 1; and (L1, A, L2) – at stage 2 (see also Table 4, #S 6).

**Case 2** (when *k*=2): T - at stage 2; (S, V) and (L1, A, L2) – at stage 1 (see also Table 3, #S 6).

Case 1 is the most relevant assignment because of the task logic: the teacher selects the teaching model first, then the algorithms to be taught and, finally, the pure technical characteristics of the teaching environment.

PROPERTY 7. The priority relation such as HL, IL, and LL is the context-based information to govern the adaption process in using domain generators and meta-generators.

Indeed, by selecting the value of the parameter T, teacher adapts the teaching model to his\her needs; by selecting the type of an algorithm (parameter A), teacher makes adaptation of the teaching content to different groups of students.

CONSTRAINT 1. The priority relation (if any) should be indicated in the meta-interface by the meta-designer using comments written before each parameter use.

This information will be used by refactoring tool (method). What will happen if there was not introduced the full list of priorities, or the priority relation has been missed at all? In the first case, a given priority (say HL, or LL) is still useful information in selecting stages (due to RULE 1) but not enough. If there is no priority at all and with regard to the fact that the parameter space is not changed in refactoring (only permutation is performed), the following question can be raised. What is better in constructing re-factorings (meaning selecting stages): either to have "more meta-generators and less generators", or "less meta-generators and more generators"? As generators are "closer to user" in the sense that they produce programs directly to integrate them into the user's system, from this perspective, the better variant is "less meta-generators and more generators". This reasoning leads to the following *heuristic* rule that specifies how many groups of parameters (presented as connected components $G_i$ ) should be allocated to each non-allocated stage remaining after applying RULE 1.

RULE 2. If, after applying RULE 1, the number of non-allocated stages $k^*<g^*$ (where $g^*$ is the number of yet non-allocated connected components), then the assignment is as follows:

(a) one connected component for each non-allocated stage, except stage 1, is allocated and
(b) $g^*- k^*+1$ connected components for stage 1 (whether or not the stage has the allocated parameters) are allocated.
(c) If $k^*=g^*$, then there is the only one group for each stage (see Fig. 5(c)).

When all parameters are allocated to stages, the refactoring process can be already performed as it is specified by the next rules.

RULE 3. If, after applying RULE 1, the number of non-allocated stages $k^*<g^*$ (where $g^*$ is the number of yet non-allocated connected components), then either calculate all possible assignments (if computational resources are not a matter), or let user define the assignments.

RULE 4. Refactoring of meta-interface is done first and then refactoring of meta-constructs within the meta-body follows. The first action is implemented by lifting parameters to allocated stages (replacing their location within the meta-interface). The second action is modelled by inserting the de-activating index before meta-functions that are related to lifted parameters.

RULE 5. De-activating index (see DEFINITION 6), is defined by Eq. (10) (also see Fig. 2(a)):

Index = 0, for stage $k$; 1, for stage ($k$-1), etc.

$$\text{and } \sum_{a=0}^{k-2} 2^a \text{, for stage 1.} \tag{10}$$

Table 1 explains the use of de-activating process and presents the index value to ensure refactoring of meta-constructs for PHP (note that (10) is also valid for Java and C++).

**Table 1.** Examples of using de-activating for PHP

| Where applied | Examples in PHP | Index value |
|---|---|---|
| In stage 3 | `$P1 = "AND";` | 0 |
| In stage 2 | `echo "\$P2 = 3;";` | 1 |
| In stage 1 | `echo "echo \"\\\$P3 = Y;\";";` | 3 |

Further we identify the use of RULES 1, 2 and 4 as *Strategy 1* to describe the refactoring method in Section 5. *Strategy 1* enables to produce the only one solution of Task 1. We use also *Strategy* 2 (RULE 3) for investigation some properties of refactoring (such as checking equivalence of transformations, complexity evaluation [39, 43]). *Strategy 2* enables to produce a set of solutions (see Tables 3 and 4). If the number $g$ is less than the "magic 7" (also known as Miller's cognitive complexity [14]), then we are able to produce and check all possible transformations (see also Eqs. (8) and (9)). Otherwise, we need to restrict the number of possible solutions.

## 5. Refactoring method and experiments

### 5.1. Step-wise description of the method

The following assumptions are accepted: i) the given meta-program or $M$ is correct; ii) the parameter space of $M$ is pre-specified in advance and cannot be changed; iii) each stage must have at least one parameter or one group of related parameters; iv) the priority-based relationship should be introduced by the domain expert or by meta-designer indicating the priority, e.g. as a comment before each assignment statement within the meta-interface of $M$ .

Let us be given the specification $M$ and the required number of stage $k_0 (k_0 > 1)$ for $M$. The method we describe below is given as a sequence of steps being supported by devised RULES to solve Task 1.

***Step 1.*** Make choice of strategy (*Strategy* 1 or *Strategy* 2); $M^k = \varnothing$.

***Step 2.*** Analyze the meta-interface of $M$ , construct the graph $G(P,U)$, its connected components $G_i$ (if any) and identify $g$.

***Step 3.*** **If** *priority relation*=***true*** **then** change the graph $G(P,U)$ into $G^*(P,U)$ by introducing priorities.

***Step 4.*** **If** $g > k_0$ **then** *go to Step 6* (meaning the solution exists) **else go to** *Step 5*.

***Step 5.*** $k_0 = k_0 - 1$ ; **go to** *Step* 4 **until** $k_0 \neq 1$; otherwise **go to** *Step 8* (meaning there is no solution).

***Step 6.*** **If** *Strategy* 1 **then do**

**If** *priority relation*=***true*** **then do**

Sort connected components $G_i$ by priority in decreasing order;

Assign the ordered connected components to stages according to RULE 1;

Identify yet non-allocated stages **end do**

**If** non-allocated stages **=true then do**

apply RULE 2;

apply RULE 4 and RULE 5 to perform refactoring; perform fulfilling $M^k$ ; **end do**;

**end if**;

***Step 7.*** **If** Strategy 2 then **do**

apply RULE 3;

for each assignment given by RULE 3,

apply RULE 4 and RULE 5 and form a set of re-factorings;

**end do**

***Step 8.*** **End**.

### 5.2. Methodology and results of experiments

The methodology we have chosen to provide experiments includes the following steps: (a) selection of application domains, target (domain) languages and meta-languages; (b) identification of the scope of experiments; (c) solving Task 2 through experiments; (d) evaluation of the experiments. We have found the following requirements relevant for step (a): domains and their languages are to be as simple as possible, but, on the other hand, they are to be related to real applications; though we have repeated our experiments using two meta-languages aiming to clarify their impact on transformation characteristics such as complexity, we present the results only for one meta-language (PHP) here. On this account, four domains under investigation were selected: abstract strings generated using alphabet {0, 1} (not presented here); test-frames based on the alphabet {0, 1, X} that are

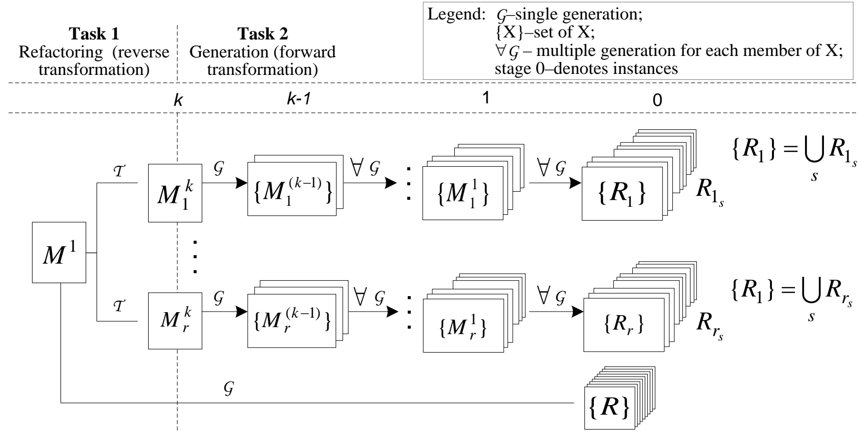


**Figure 6**. *k*-stage transformation/generation processes within the same meta-parameter space

used in hardware testing to compact input vectors [11]; the so-called *L*-systems that play extremely important role in designing interfaces of systems [28]; CS topics (learning programs) for educational robots [40]. All selected domains (except the first) are characterized by the great degree of variability.

Formally, *L*-systems [28] are a string-rewriting grammar expressed as an ordered triplet (*U, ω, H*), where: *U* – alphabet, i.e., a set of symbols (variables) that can be replaced by other symbols; *ω* – start symbol, also called axiom or initiator, defining the initial state of the system ($\omega \in U$); *H* – set of production rules defining the way variables can be replaced by a combination of other variables iteratively ($H \subset U \times U^*$).

Program generation (see Task 2 to the right of the dotted line in Fig. 6) is based on the *multi-level* forward transformations processes. These processes are supported by PHP processor. They are completely automatic.

The aim of experiments is to show that using the method (see sub-section 5.1) for *different domains* the following property holds:

$$\forall_{k=2,3}(\forall_{i(i\in[1,r])}(\{R_i\}=\{R\})) \text{ is true,} \qquad (11)$$

where $R_i = \bigcup_s R_{i_s}$ (see also Fig. 6); $R$ is the set of all instances derived from $M^1$; $R_{i_s}$ is the set of all instances derived from $M_i^k$ . Figure 6 outlines the schematic view of experiments we have carried out (see also the mode *Strategy 2* within the method). Stage 0 presents overall instances derived from different branches (meta-programs).

The result of solving Task 1 is a set of all single *k*-stage meta-programs (from 1 to *r*), where 1 means some *k*-stage meta-program with one parameter and *r* – the number of all possible permutations of parameters at stage *k*.

The relationship, say, $\{R_r\} = \bigcup_s R_{r_s}$ means the writing (concatenation) of generation results into the same file numbered by *r*.

We present results in Tables 2-4 below. Table 2 gives some characteristics of $M^1$ as data to solve Task 1. Note that dependent parameters and their values are given in parentheses (see, e.g. (2, 3, 4) and (4, 4, 6) in line S# 6 of Table 2 and sub-section 4.4).

Table 3 presents results (Task 1 and Task 2) of all possible transformations $M^1$ into $M_i^k (i \in [1,r])$ and generation when *k*=2. Table 4 presents results of the same transformations and generation when *k*=3. As it is clear from comparison of the number of generated instances given in Tables 2-4, this number is the same. Note that this property is not enough to approve equivalence of such transformations. We have also checked the coincidence of the content of instances within all files formally interpreted here as {*R*}, {*R$_1$*},…, {*R$_r$*} (see Fig. 6) using the *Excel* facilities (Conditional Formatting) for checking the identity of the files. In all such cases the full identity of instances within files was obtained (the order of instances within file was different only). Therefore, we were able to conclude that the equivalence condition (Eq. (11)) holds, meaning that the transformations $M^1 \xrightarrow{T} M^2 \xrightarrow{T} M^3$ are semantics preserving structural transformations, or refactoring. Note that the partial results ( $M^1 \xrightarrow{T} M^2$ ) for other domains using a slightly different methodology were described in [39, 43].

Tables 2-4 contain also the results of using the method, when *Strategy 1* and priority relations were applied, to define *a concrete* refactoring for solving real tasks. Priority relations are helpful to identify how to assign parameters (or groups of parameters represented as graphs $G^*(P,U)$ in Fig. 5) to stages. For example, we have presented the results of applying *Strategy* 1 in columns 4-6 (see Tables 2-4).

As we could not be able to reveal semantics of parameters in Tables (parameters and their values are expressed by abstract numbers here), we recommend (for clearness) to connect the result of Line follower 2 (see sample 6 given in bold in Tables 3 and 4) with the

**Table 2.** Characteristics of 1-stage meta-programs $M^1$

| S# | Name of MP | # of meta-para-meters | Dependent meta-parameters given by numbers | # of values (for each parameter) | Priority relation | # of generated instances $\lvert R\rvert$ $(M^1 \xrightarrow{G} \{R\})$ |
|---|---|---|---|---|---|---|
| 1 | L-system | 4 | No | (5,4,4,4) | 3-null; 1- LL | 320 =5*4*4*4 |
| 2 | Test frame | 5 | No | (2,2,2,2,2) | 5-null | 32=2*2*2*2*2 |
| 3 | Calibration of NXT robot | 4 | No | (3,4,3,3) | 1-HL; 3- null | 108=3*4*3*3 |
| 4 | Calibration of NXT robot with context | 5 | No | (2,3,4,3,3) | 1-HL; 1-IL; 3-LL | 216=2*3*4*3*3 |
| 5 | Line follower 1 | 5 | 2,3 | (2,(4,20),3,4) | 1-HL; 2-IL; 2-LL | 960=2*(2*10+2*10)*3*4 |
| 6 | **Line follower 2** | **6** | **(2,3,4)** | **(2,(4,4,6),3,3)** | **1-HL; 3-IL; 1-LL;1-null** | **360=2*(2*4+2*6)*3*3** |
| 7 | Ornament's design | 3 | No | (2,3,2) | 3-null | 12=2*3*2 |

**Table 3.** Characteristics of $M^1$ refactoring into *k*-stage meta-programs when *k*=2 (samples 1&2 are omitted since they are in [39])

| S# | Name of MP | Total # $M^2$ derived from $M^1$ | # of meta-parameters of $M^2$ | # of meta-parameters of $M^1$ | Total # of generated $M^1$ from all $M^2$ | Total # of generated Instances from $M^1$ | Equivalency condition (see Eq.(11)) |
|---|---|---|---|---|---|---|---|
| 3 | Calibration of NXT robot | 7 | 1-HL | 3-null | 3 | 108=3*(4*3*3) | **true** |
| | | | 1-HL, 1-null | 2-null | 12 =3*4 | 108=12*(3*3) | **true** |
| 4 | Calibration of NXT robot with context | 2 | 1-HL | 1-IL, 3-LL | 2 | 216=2*(3*4*3*3) | **true** |
| 5 | Line follower 1 | 2 | 1-HL | 2-IL, 2-LL | 2 | 960 =2*((2*10+2*10)*3*4) | **true** |
| 6 | **Line follower 2** | **2** | **1-HL** | **3-IL, 1-LL, 1-null** | **2** | **360 =2*(2*4+2*6)*3*3** | **true** |
| 7 | Ornament's design | 6 | 1-null | 2-null | 2 | 12=2*(3*2) | **true** |
| | | | 2-null | 1-null | 6=2*3 | 12=6*(2) | **true** |

**Table 4.** Characteristics of $M^1$ refactoring into *k*-stage meta-programs when *k*=3

| S# | Name of MP | Total # $M^3$ derived from $M^1$ | # of meta-para-meters of $M^3$ | # of meta-para-meters of $M^2$ | # of meta-parameters of $M^1$ | Total # of generated $M^2$ from all $M^3$ | Total # of generated $M^1$ from all $M^2$ | Total # of generated instances from all $M^1$ | Equivalency condition (see Eq. (11)) |
|---|---|---|---|---|---|---|---|---|---|
| 1 | L-system | 12 | 1-null | 1-null | 1-null, 1-LL | 5 | 20=4*4 | 320=20*(4*4) | **true** |
| | | | 2-null | 1-null | 1-LL | 20=5*4 | 80=20*(4) | 320=80*(4) | **true** |
| 2 | Test frame | 150 | 1-null | 2-null | 2-null | 2 | 8=2*(2*2) | 32=8*(2*2) | **true** |
| | | | 2-null | 1-null | 2-null | 4=2*2 | 8=4*(2) | 32=8*(2*2) | **true** |
| 3 | Calibration of NXT robot | 12 | 1(HL) | 1-null | 2-null | 3 | 12=3*(4) | 108=12*(3*3) | **true** |
| | | | 1(HL) | 2-null | 1-null | 3 | 36=3*(4*3) | 108=36*(3) | **true** |
| 4 | Calibration of NXT robot with context | 1 | 1(HL) | 1(IL) | 3-LL | 2 | 6=2*(3) | 216=6*(4*3*3) | **true** |
| 5 | Line follower 1 | 1 | 1(HL) | 2(IL) | 2-LL | 2 | 80= 2*(2*10+2*10) | 960=80*(3*4) | **true** |
| 6 | **Line follower 2** | **1** | **1(HL)** | **3(IL)** | **1-LL, 1-null** | **2** | **40= 2*(2*4+2*6)** | **360=40*3*3** | **true** |
| 7 | Ornament's design | 6 | 1-null | 1-null | 1-null | 2 | 6=2*(3) | 12=6*(2) | **true** |
| | | | 1-null | 1-null | 1-null | 3 | 6=3*(2) | 12=6*(2) | **true** |

S – sample; MP – meta-program; $M^1$ - 1-stage meta-program; $M^2$ - 2-stage meta-program; $M^3$ - 3-stage meta-program; HL – highest level; IL –intermediate level; LL –lowest level: null – no priority.

description given in sub-section 4.4 (Case 2 and Case 1) and Fig. 5.

As we could not be able to reveal semantics of parameters in Tables (parameters and their values are expressed by abstract numbers here), we recommend (for clearness) to connect the result of Line follower 2 (see sample 6 given in bold in Tables 3 and 4) with the description given in sub-section 4.4 (Case 2 and Case 1) and Fig. 5.

We summarize the experimental results as follows: 1) the hypothesis that those transformations ($M^1$ into $M^2$ or $M^3$) are actually refactoring is true; 2) there is a large enough space for selecting program instances for adaptation for a concrete context of use; 3) the priority–based information is helpful for this adaptation in solving real world tasks.

## 6. Discussion and evaluation

The necessity of refactoring to transform the given correct heterogeneous meta-program into its *k*-stage format arises due to practical and theoretical reasons. The practical needs have come from our extensive experiments in using NXT educational robots [30] in the real teaching setting (school) to teach CS topics. Aiming at the increase of efficiency and flexibility in the content preparation and continuous changes (by both teacher and students), we describe the content as meta-programs. Because of the extremely wide e-learning context (social, pedagogical, technological characteristics of Robots, tasks specificity, etc.), meta-programs may contain a large number of parameters. Due to the necessity of managing changeability and adaptation of the teaching content to the context of use (we do that using context-based priority relation here), we have found refactoring of meta-programs into the *k*-stage meta-programs as a relevant and beneficial technology. The practical benefits are not restricted by one domain. We have obtained the similar observation in other domains of great importance, such e-commerce [21, 44].

The basic results of the paper can be summarized as follows.

1. The theoretical background introduced and experiments we have carried out approved the hypothesis that the meta-program refactoring into *k*-stage meta-program is the semantics preserving transformation.

2. Though we have not considered the development of the meta-program here (it was given as input in describing the proposed method), the concept of staging is also useful to better understanding the development process due to the possibility of its systemizing. For example, in developing a meta-program, a designer is able to introduce parameters into the specification gradually, in stages, making testing after each stage, thus, in this way, simplifying the procedure.

3. The key idea of the method to transform a meta-program into its *k*-stage structure is based on the de-activation/activation mechanism (process) to de-activate/activate the adequate constructs at the suitable stage of the given specification. A meta-language must have the constructs (features) to ensure the realization of the mechanism.

4. The benefits of the approach are: a) it provides a theoretical background to develop meta-program refactoring tools; b) it enables to construct a set of the lower-level generators that are derived from the *k*-stage meta-program; c) it extends the known multi-stage programming concept applying it in another context of use, i.e. in the heterogeneous meta-programming domain; d) it contributes to better understandability of heterogeneous meta-programming domain; e) it extends the generative reuse, though in a narrow and specific way.

5. The method has some limitations too. First, its use is restricted by heterogeneous meta-programs only. Second, there are some difficulties in applying de-activating index when the number of stages is more than 3. The reason is the significant decrease of readability of such meta-specifications (meta-designer's view) because of the abnormal growth of the de-activating index value (e.g. when $k$ = 4, index = 7 for such languages as PHP, Java, C++). However, the refactoring tool hides (eliminates) this deficiency. Finally, it is difficult to form the precise criteria for refactoring due to the task complexity and context dependency upon the application task.

## 7. Conclusion

1. The graph-based approach has been found as a relevant basis to theoretically approving the proposed method to solve the meta-program refactoring problem. Both theoretical and practical results obtained provide sufficient information to build refactoring tools.

2. Though we have obtained the conditions of resolving the problem in general case and we have identified the upper bound on the needed number of stages to refactoring a heterogeneous meta-program into the *k*-stage format, the refactoring process cannot be completely automatic, if the context of refactoring is not fully described.

3. As the refactoring context is highly dependent on the application domain, in the case of the partial description of the context information, the method can ensure automatic refactoring by providing all possible variants, if the cognitive complexity does not exceed the boundary of the 'magic 7 problem'. This bound has been selected to reduce/save computational resources in our experiments.

4. From the user's perspective, refactoring raises the abstraction level of transformations because, at a higher stage, he/she uses a less amount of information, which is presented in the user-friendly format. From the meta-designer's perspective, refactoring-based

transformation preserves approximately the same abstraction level.

## References


[1] **A. Cisternino.** Multi-Stage and Meta-Programming Support in Strongly Typed Execution Engines, *PhD Thesis,* 2003. http://phd.di.unipi.it/theses/phdthesis_cisternino.pdf

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

[3] **A. Ortiz.** An introduction to metaprogramming. *Linux Journal,* 2007, Vol. 158, 1-6.

[4] **B. Boehm.** Some Future Trends and Implications for Systems and Software Engineering Processes. *Systems Engineering*, 2006, Vol. 9, No. 1, 12-29.

[5] **B. D. Bois, P. Van Gorp, A. Amsel, N. Van Eetvelde, H. Stenten, S. Demeyer, T. Mens**. A Discussion on Refactoring in Reseaerch and Practice. *Technical Report, University of Antwerp*, 2004.

[6] **C. Birtolo, D. De Chiara, M. Ascione, R. Armenise.** A Generative Approach to Product Bundling in the e-Commerce Domain. In: *The 3rd World Congress on Nature and Biologically Inspired Computing – NaBIC*, Spain, 2011, pp. 169-175.

[7] **D. Batory**. Feature Models, Grammars, and Propositional Formulas. In H. Obbink and K. Pohl (eds.). *9th Int. Software Product Line Conf.*, LNCS 3714, Springer, 2005, pp. 7-20.

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

[9] **D. Batory.** Program Refactoring, Program Synthesis, and Model-Driven Development. *Invited Presentation at European Joint Conferences on Software Theory and Practice of Software (ETAPS) Compiler Construction Conference*, 2007, pp. 3-12.

[10] **D. Thomas.** Refactoring as Meta Programming? *Journal of Object Technology*, 2005, Vol. 4, No. 1, 7–12.

[11] **E. Bareiša, V. Jusas, K. Motiejūnas, R. Šeinauskas.** Functional test generation remote tool. In: *8th Euromicro Conference on Digital System Design* (DSD'05), 2005, pp. 192-195.

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

[13] **E. Visser.** A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 2005, Vol. 40, No. 1, 831-873.

[14] **G. Miller.** The Magic Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *The Psychological Review*, March, 1956.

[15] **H. A. de Jong, P. A Olivier**. Generation of abstract programming interfaces from syntax definitions. *The Journal of Logic and Algebraic Programming*, 2004, Vol. 59, No. 1, 35-61.

[16] **J. Eckhardt, R. Kaiabachev, E. Pasalic, K. Swadi, W. Taha.** Implicitly Heterogeneous Multi-Stage Programming for FPGAs. *Journal of Computational Information Systems*, 2010, Vol. 6, No. 14, 4915-4922.

[17] **J. A. Gray.** Toeing the Line: Experiments with Line-following Algorithms. *Technical Report,* 2003. http://www.fll-freak.com/misc/01-jgray_report.pdf.

[18] **J. R. Cordy, M. S. Sarkar.** Metaprogram Implementation by Second Order Source Transformation. *Workshop at Generative Programming and Component Engineering Conference* (GPCE'04), Vancouver, Canada, 2004, pp. 5-6.

[19] **K. Pohl, G Böckle, F. J. van der Linden.** Software Product Line Engineering: Foundations, Principles and Techniques. New York, Inc., Secaucus, NJ, Springer-Verlag, 2005.

[20] **K. C. Kang, J. Lee, P. Donohoe.** Feature-Oriented Product Line Engineering. *IEEE Software*, 2002, Vol. 19, No. 4, 58-65.

[21] **K. Valinčius, V. Štuikys, R. Damaševičius.** Understanding of E-commerce IS through Feature Models and Their Metrics. In: *IADIS International Conference Information Systems,* 13-15 March, Lisbon, Portugal, 2013, pp. 55-62.

[22] **M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts.** Refactoring: Improving the Design of Existing Code, Addison Wesley. 1999. http://www.refactoring.com.

[23] **M. G. J. van den Brand, P. E. Moreau, J. J. Vinju.** A generator of efficient strongly typed abstract syntax trees in Java. *IEE Proceedings - Software*, 2005, pp. 70-78.

[24] **MDA**. OMG Model Driven Architecture. www.omg.org/mda.

[25] **M. Zhang, N. Baddoo, P. Wernick, T. Hall**. Improving the Precision of Fowler's Definitions of Bad Smells. *32nd Annual IEEE Software Engineering Workshop,* IEEE, 2009, pp. 161-166.

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

[27] NXT User Guide. http://cache.lego.com.

[28] **P. Prusinkiewicz, A. Lindenmayer.** The Algorithmic Beauty of Plants. New York, Springer-Verlag, 1990.

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

[30] **R. Burbaitė, V. Štuikys, R. Marcinkevicius.** The LEGO NXT Robot-based e-Learning Environment to Teach Computer Science Topics. *Electronics and Electrical Engineering*, 2012, Vol. 18, No. 9, 113-116.

[31] **S. Trujillo, D. Batory, O. Diaz.** Feature Refactoring a Multi-Representation Program into a Product Line. *Proceedings of the 5th international conference on Generative programming and component engineering,* New York, NY, 2006, pp. 191-200.

[32] **S. Trujillo, M. Azanza, O. Díaz.** Generative Metaprogramming. *Proc. of 6th Int. Conf. on Generative Programming and Component Eng.* (GPCE 2007), Salzburg, Austria, October 1-3, 2007, pp. 105–114.

[33] **T. Mens, K. Czarnecki, P. Van Gorp.** A Taxonomy of Model Transformations. *Electronic Notes in Theoretical Computer Science*, 2006, Vol. 152, 125–142.

[34] **T. Mens, T. Tourw.** A Survey of Software Refactoring. *IEEE Transactions on Software. Engineering*, 2004, Vol. 30, No. 2, 126-139.

[35] **T. Sheard.** Accomplishments and Research Challenges in Meta-Programming. *Proc. of 2nd Int. Workshop on Semantics, Application, and Implementation of Program Generation (SAIG'2001),* Florence, Italy. LNCS 2196, Springer, 2001, pp. 2-44.

[36] **T. Tourwe, T. Mens.** Identifying Refactoring Opportunities Using Logic Meta Programming. In: *Software Maintenance and Reengineering,* 2003. *Proceedings. Seventh European Conference on. IEEE,* 2003, pp. 91-100.

[37] **T. L. Veldhuizen.** Tradeoffs in Metaprogramming. Proc. *In: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation.* ACM, 2006, pp. 150-159.

[38] **V. L. Winter.** Program Transformation: What, How and Why. In: Wah, B.W. (Ed.): *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc. 2004.

[39] **V. Štuikys, K. Bespalova.** Methodology and Experiments to Transform Heterogeneous Meta-Program into Meta-Meta-Programs. In: *The 18th international conference on information and software technologies* (ICIST 2012). September 13-14, Kaunas, Lithuania, 2012, pp. 210-225.

[40] **V. Štuikys, R. Burbaitė, R. Damaševičius.** Teaching of Computer Science topics using meta-programming-based GLOs and LEGO robots. *Informatics in Education,* 2013, Vol. 12, No. 1, 125-142.

[41] **V. Štuikys, R. Burbaite.** Two-Stage Generative Learning Objects. *Information and Software Technologies, Communications in Computer and Information Science*, 2012, Vol. 319, 332-347.

[42] **V. Štuikys, R. Damaševičius, G. Ziberkas, K. Valinčius.** Understanding of heterogeneous multi-stage meta-programs. *Information Technology and Control,* 2012, Vol. 41, No. 1, 23-32.

[43] **V. Štuikys, R. Damaševičius.** Equivalent Transformations of Heterogeneous Meta-Programs. *Informatica*, 2013, Vol. 24, No. 2, 315-337.

[44] **[V. Štuikys, R. Damaševičius.** Meta-Programming and Model-Driven Meta-Program Development: Principles, Processes and Techniques. Springer, 2013.

[45] **V. Štuikys, R. Damaševičius.** Meta-programming Techniques for Designing Embedded Components for Ambient Intelligence. In: T. Basten, M. Geilen, H. de Groot (eds.), *Ambient Intelligence: Impact on Embedded System Design*. Kluwer Academic Publishers, 2003, pp. 229-250.

[46] **W. Löwe, M. Noga.** Metaprogramming Applied to Web Component Deployment. *Electronic Notes in Theoretical Computer Science,* 2002, Vol. 65, No. 4, 106-116.

[47] **W. Taha.** A Gentle Introduction to Multi-stage Programming. *Domain-Specific Program Generation, Lecture Notes in Computer Science*, 2004, Vol. 3016, 30-50.

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

Received March 2013.