KAUNAS UNIVERSITY OF TECHNOLOGY

KRISTINA BESPALOVA

AUTOMATED META-PROGRAM DEVELOPMENT AND SPECIALIZATION USING FEATURE-BASED MODEL TRANSFORMATIONS

Summary of Doctoral Dissertation

Physical Sciences, Informatics (09P)

2015, Kaunas

The dissertation was prepared at Kaunas University of Technology, Faculty of Informatics, Department of Software Engineering during 2011–2015.

Scientific supervisor: Prof. Dr. Habil. Vytautas ŠTUIKYS (Kaunas University of Technology, Physical Sciences, Informatics – 09P).

Dissertation Defense Board of Informatics Science Field:

Prof. Dr. Habil. Rimantas BARAUSKAS (Kaunas University of Technology, Physical Sciences, Informatics – 09P) – **chairman**;

Prof. Dr. Andrej BRODNIK (University of Ljubljana, Physical Science, Informatics – 09P)

Prof. Dr. Valentina DAGIENĖ (Vilnius University, Technological Sciences, Informatics Engineering – 07T);

Prof. Dr. Vacius JUSAS (Kaunas University of Technology, Physical Sciences, Informatics – 09P);

Prof. Dr. Alfonsas MISEVIČIUS (Kaunas University of Technology, Physical Sciences, Informatics – 09P).

The official defence of the Dissertation will be held at the open meeting of the Board of Informatics Science Field at 11 a. m. on December 14, 2015 in the Dissertation Defence Hall of the Central Building of Kaunas University of Technology.

Address: K. Donelaičio g. 73-403, LT–44249, Kaunas, Lithuania.
Phone (370) 37 30 00 42, fax. (370) 37 32 41 44, e-mail doktorantura@ktu.lt

The send out date of the summary of the Dissertation is on 13 November 2015.

The Dissertation is available at http://ktu.edu/lt and at the Library of Kaunas University of Technology (K. Donelaičio g. 20, Kaunas, Lithuania).

KAUNO TECHNOLOGIJOS UNIVERSITETAS

KRISTINA BESPALOVA

AUTOMATIZUOTAS METAPROGRAMŲ KŪRIMAS IR SPECIALIZAVIMAS PANAUDOJANT POŽYMIAIS GRINDŽIAMŲ MODELIŲ TRANSFORMACIJAS

Daktaro disertacijos santrauka

Fiziniai mokslai, Informatika (09P)

2015, Kaunas

Disertacija rengta 2011-2015 m. Kauno technologijos universiteto Informatikos fakultete, Programų inžinerijos katedroje.

Mokslinis vadovas: Prof. habil. dr. Vytautas ŠTUIKYS (Kauno technologijos universitetas, fiziniai mokslai, informatika – 09P).

Informatikos mokslo krypties daktaro disertacijos gynimo taryba:

Prof. habil. dr. Rimantas BARAUSKAS (Kauno technologijos universitetas, fiziniai mokslai, informatika – 09P) – **pirmininkas**;

Prof. dr. Andrej BRODNIK (Liublijanos universitetas, fiziniai mokslai, informatika – 09P)

Prof. dr. Valentina DAGIENĖ (Vilniaus universitetas, technologijos mokslai, informatikos inžinerija – 07T);

Prof. dr. Vacius JUSAS (Kauno technologijos universitetas, fiziniai mokslai, informatika – 09P);

Prof. dr. Alfonsas MISEVIČIUS (Kauno technologijos universitetas, fiziniai mokslai, informatika – 09P).

Disertacija bus ginama viešame Informatikos mokslo krypties tarybos posėdyje, kuris įvyks 2015 m. gruodžio 14 d. 11 val. Kauno technologijos universiteto centrinių rūmų disertacijų gynimo salėje.

Adresas: K. Donelaičio g. 73-403, LT-44249, Kaunas, Lietuva.
Tel. (+370) 37 30 00 42, faksas (+370) 37 32 41 44, el. paštas
doktorantura@ktu.lt

Disertacijos santrauka išsiųsta 2015 m. lapkričio 13 d.

Disertaciją galima peržiūrėti interneto svetainėje http://ktu.edu/lt ir Kauno technologijos universiteto bibliotekoje (K. Donelaičio g. 20, Kaunas).

1. INTRODUCTION

1.1. Relevance of the topic

Transformation of one form objects (process, energy products and etc.) to another is an essential attribute of all technical systems. In informatics, this attribute is even more important for the following reasons: (1) The transformation objects are not physical objects, but their abstract representations (data, applications and models); (2) there are many forms of representation; (3) abstract representations enable the implementation of transformations much easier; (4) transformation in informatics determines practically any kind of computer system functionality.

The transformations within systems are used in different contexts and cover a wide spectrum of processes, from the lowest level to the highest. The lowest level is the traditional transformation: the processor, operating system. A higher level transformation is a compilation, even higher – application design transformation, and the highest – system-system transformation. In software engineering and informatics, the main transformation is performed by programs and models. The program transformation is used in a wide range of applications, including compiler construction, optimization, program synthesis, transformation, software renovation, and reverse engineering (Visser, 2001).

Research in the field of model and program transformation is very wide, but all kinds of transformations seek the same goal – to increase productivity and efficiency in system development. Therefore, the main goal of transformation is automation.

Over the last decade, a striking leap in information technology advances surpassed can be observed. For example, the base technological advancements have all expectations. Today people are using new technology elsewhere. In other words, we are living and working in the digital world, where changes are a constant phenomenon. With the development of information technology (IT), the following trend is evident: an extremely rapid growth of IT-based systems and the ever-increasing requirements imposed by market pressure. On the other hand, the software content within the systems is growing too, even at a higher rate than the systems do. It is especially true in the sectors of embedded systems and web applications (e.g. the Internet of Things).

Rapid technological advances have a direct impact on program size, quality and complexity. These attributes provide great challenges for the system designers. The approved way to respond to the challenges is the use of automated design methods, supported by transformation tools. The development of contemporary systems is based on the reuse methodology. This methodology is based on the concept of product lines (it can be seen as a meta-system) also known as Product Line Engineering (PLE) (Pohl, Böckle and van der Linden, 2005). The latter covers the domain analysis, modelling (creation of models and meta-models) and the development of generic (meta-) components and program

generators. The PLE methodology focuses on reuse and mainly operates with feature-based models. The methodology is represented by two levels (domain engineering and application engineering).It promotes the whole development process, uses the high-level models, abstractions and transformations in order to achieve a higher degree of reuse and automation. It was found in recent studies (through analysis) that the model-based methodology is prevailing. However, there are still many unresolved problems associated with analysis, variability modelling (Capilla, Bosch and Kang, 2013), mapping, transformation and realization (Biehl, 2010; Fioravanti et al. 2011; Völteret al., 2013; Zhang, 2014).

Therefore, this dissertation deals with specific tasks that so far have not been explored sufficiently: feature model transformation into the heterogeneous meta-program and the transformation of the meta-program itself, e.g. aiming at its specialization and adaptation. The heterogeneous meta-program development is a complex process that requires both a deep knowledge and tool support, which is the main topic of the dissertation. By the heterogeneous meta-program (further meta-program) throughout the dissertation it is meant the one, which is described using two languages: meta-language and target language. The latter specifies the base domain functionality. The first is used for generalization, i.e. for expressing the domain variability through parameterization (Štuikys and Damaševičius, 2013).

1.2. Research object

In this dissertation, the research object is the problem domain feature models, meta-programs, their development and transformation processes and methods.

1.3. Research objective

The objective of the research is to create and explore the methodology for the meta-program automated creation and transformation, including the tools that support the processes.

1.4. Research tasks

1. Analysis and evaluation of the methodologies related to model and program (meta-program) transformations.
2. Creation of a meta-program using the feature model transformation.
3. The initial meta-program automatic transformation (specialization) into the multi-stage meta-program[1] aiming at its adaptation.
4. Development and research of the meta-program design processes, transformation algorithms and corresponding tools.

[1] Multi-stage meta-program is a lower-level meta-program generator. It is designed so that to enable the execution process in separate stages sequentially. A stage is defined by a subset of the *active* parameters, while the remaining is being *deactivated.*

1.5. Statements presented for defence

5. Problem and solution domain feature-based models enable firstly to systematize and then to create meta-programs semi-automatically.

6. Formal models of meta-program specialization and adaptation ensure the functionality and correctness of the transformation tool.

7. The transformation tools developed ensure efficient meta-program creation, transformation and support.

1.6. Scientific novelty

1. The proposed method for developing meta-programs is based on the feature model transformations, thus enabling the automation of the process.

2. The established multi-stage transformation condition (i) for the *existence of solutions* and (ii) for the *permissible number of stages*. Both enable the *generalisation* of the two-stage meta-program transformation into the multi-stage meta-program.

3. Proposed the *complete* meta-program design process, comprising: (i) model and meta-program creation (using the design tool), (ii) their transformations (using the refactoring tool), and (iii) customization and generation / adaptation of a target program.

1.7. Practical relevance

1. Automated meta-program design (tool) was used for the educational robot control program generation.

2. Automated multi-stage meta-program design (called refactoring tool) was used for context-aware automatic adaptation of the meta-programs used in the real teaching setting.

1.8. Approbation of the research results

The main results of the dissertation are represented in 8 scientific publications: 2 in the periodical scientific journals (ISI Web of Science) and 6 in international conference proceedings.

1.9. The structure and volume of the dissertation

The dissertation consists of an introduction, 6 main chapters and conclusions. A list of author's publications, a list of references and 4 appendixes are also given. The total volume of the dissertation consists of 153 pages, including 48 figures, 20 tables and 217 references.

2. MODELS AND PROGRAM TRANSFORMATION METHODS

Analysis and manipulation of a program source code are regarded as one of the most important computing aspects (Harman, 2010). As the size and complexity of software is continuously growing, the manual manipulation becomes ever more infeasible. At present, however, there is an evident shift from the program code transformation towards program model transformation.

In this dissertation, feature-based modelling has been adopted to build the tool for the semi-automatic development of the meta-program. Meta-programming is a higher-level programming paradigm that deals with the methodology of manipulating programs as data (Štuikys and Damaševičius, 2013). The result of the manipulation is the lower-level program. The concept of meta-programming has been introduced to support, to enhance and to enforce reusability in the domain. With respect to reusability in mind, the transformation processes should be handled and managed as effectively as possible. Here, for this purpose, the reuse-based framework borrowed from the SWE domain has been introduced and applied, which is known as design-for-reuse (D*f*R) and design-with-reuse (D*w*R) (Sametinger, 1997). In Fig. 2.1, a research framework that gives a general understanding of the proposed approach is presented.

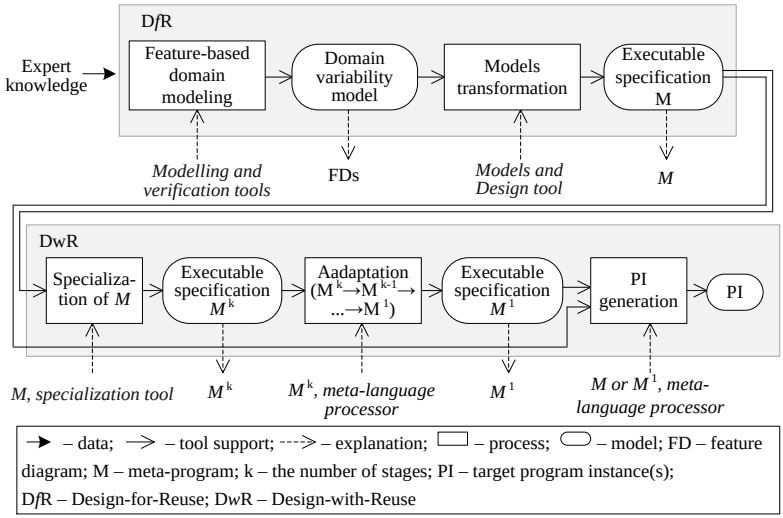

Fig. 2.1 A general research framework

D*f*R includes the processes of domain modelling and the processes of meta-program design. To model the domain (which is highly heterogeneous), the system uses the expert's knowledge, the known domain analysis methods resulting in the creation of feature models to design a meta-program. The designed meta-program, in fact, represents a family of target program instances (similarly to program families in Product Line Engineering (Pohl et al., 2005).

D*w*R includes the processes of meta-program specialization for the adaptation and generation of target program instances as the domain content is derived automatically from the meta-program specification using the meta-language processor. As a result, the user is able to create a multi-stage meta-program that can be adapted to produce various variants of use on demand.

3. HETEROGENEOUS META-PROGRAM DEVELOPMENT USING FEATUDE MODEL TRANSFORMATION

A meta-program is a program generator that generates other programs or program parts. Meta-programming is writing of meta-programs. 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 the concrete 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 the transformations according to the pre-scribed requirements for change.

Meta-program creation is a complex task. Abstractly, building a meta-program is a process of mapping of the given problem domain onto the solution domain. Formally, it is expressed as:

$$SR = PD \rightarrow SD, \tag{3.1}$$

here, *SR* – solution result, *PD* – problem domain and *SD* – solution domain.

Hereby, the problem domain means a domain model that is to be implemented using heterogeneous meta-programming. The solution domain means meta-programming techniques and approaches. Each domain has to be represented using the same formalism in order to make the model transformation feasible. Therefore, both domains are represented by feature models. In Fig. 3.1, the Y-chart shows a hierarchy of models for each domain.

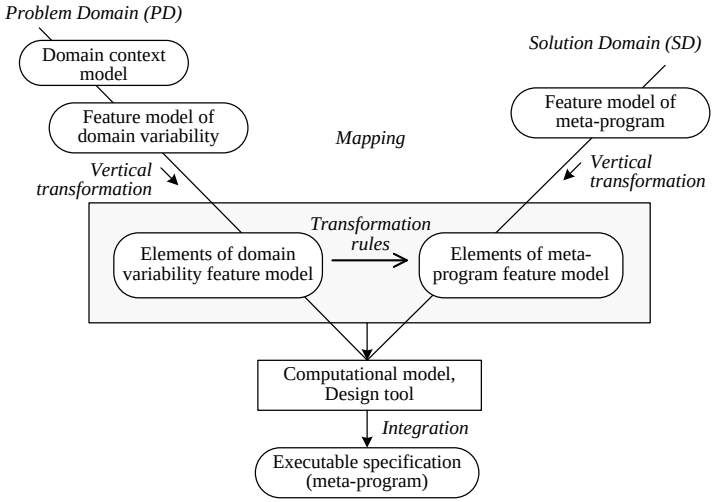

Fig. 3.1 Meta-program development framework

Judging meta-program development tasks, the problem domain and solution domain models and their transformation are individually analysed first. For both domains, the vertical transformation is applied, excluding elements of the horizontal transformation, i.e. the essential features in the feature models. Both domain models are therefore defined formally. The model formalisms enable to derive the transformation rules that should describe how problem domain elements are transformed into solution domain elements.

3.1. Fundamentals of feature models

A feature diagram is a graphical notation for feature models, represented as a tree-like or directed acyclic graph (Štuikys and Damaševičius, 2013). Feature diagrams as a domain model enable the structural, functional and behavioural variability to be expressed in a unified way using feature types and relationships.

Definition 3.1. *Variant poin*t is the parent feature whose child is grouped alternative or optional features.

Definition 3.2. *Variant* is the value of the variant point.

Definition 3.3. *Base* domain feature model is the compound: $FM = \langle G, E_{mand}, G_{xor}, G_{or}, REQ, EX \rangle$, where $G = (F, E, r)$ is a rooted tree, F is a finite set of features, $E \subseteq F \times F$ is a finite set of edges, $r \in F$ is the root feature; $E_{mand} \subseteq E$ is a set of edges that define mandatory features with their parents; $G_{xor} \subseteq P(F) \times F$, $G_{or} \subseteq P(F) \times F$, define alternative and optional feature groups and are sets of pairs of child features together with their common parent feature; *REQ* and *EX* are finite sets of constraints *requires* and *excludes* (adopted from (Acher et al., 2013)).

Definition 3.4. Problem domain feature model is a high granularity model that features detail the domain to the level of its elements.

Definition 3.5. Context feature model is the model of fuzzy variables that are treated as features taken from the set {HP, IP, LP} along with adequate constraints of the type *requires*, where: HP – High Priority, IP – Intermediate Priority, LP – Low Priority.

Note that priorities are defined in the analysis phase by a domain expert. In fact, fuzzy variables are parameter weights that are helpful to sequencing parameters in constructing the MP interface.

Definition 3.5. Extended domain feature model is the aggregation of the base feature and priority models. Formally, it is expressed as:

$$FM_P = FM_K \oplus FM_C, \tag{3.2}$$

here FM_P – extended domain feature model, FM_K – problem domain feature model, FM_C – context feature model; $\oplus$ – aggregation operator.

3.2. Fundamentals of meta-programs

The heterogeneous meta-program is the higher-level executable specification, which is coded using *at least* two languages (meta- and target) to specify and generate a set of the target program instances.

Definition 3.6. Meta-program model $\mu(M)$ is the structure: $\mu(M) = \mu(M_I) \bigcup \mu(M_B)$, where $\mu(M_I)$ – meta-interface model and $\mu(M_B)$ – meta-body model (Fig. 3.2).

Meta-interface of Meta-program: *Metadata supplied to meta-body to initiate the functioning of meta-program*
Meta-body of Meta-program: *Describing the implementation of Meta-program functionality; structurally, Meta-program specifies a set of target program instances*

Fig. 3.2 Structural model of Meta-program

Definition 3.7. In terms of the set-based notion, interface model $\mu(M_I)$ is the *n*-dimensional non-empty space of parameters and their values defined as: $\mu(M_I) = \{P;V\}$, where P – the full set of *n* parameter names, i.e. $n = |P|$, V – the ordered set of *all* parameter values.

As each parameter $P_i (P_i \in P)$ has its own set of values as follows: $\{v_{i_1}, v_{i_2}, ..., v_{i_q}\} \subset V$. Thus, we can write: $P_i := V_i = \{v_{i_1}, v_{i_2}, ..., v_{i_q}\} \in V$, i_q – the number of values of a parameter P_i. The symbol "$:=$" means 'is defined'.

Definition 3.8. Two parameters P_i and P_j $(P_i, P_j \subseteq P (i \neq j))$ are said to be *dependent upon the choice of their values* if a pair of values exists (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 $t \in [1, j_m]$; *q, m* – the number of values adequately) such that the following condition holds:

$$(v_{i_d} \text{ requires } v_{j_t}) \text{ or } (v_{i_d} \text{ excludes } v_{j_t}) = \textbf{true}. \quad (3.3)$$

Definition 3.9. Two 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 (otherwise not interacting)* if a pair of values exists (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 $t \in [1, j_m]$; *q, m* – the number of values adequately) such that the following condition holds:

$$(v_{i_d} \text{ requires } v_{j_t}) \text{ or } (v_{i_d} \text{ excludes } v_{j_t}) = \textbf{false}. \quad (3.4)$$

Definition 3.10. The graph $G(P^w, U)$ is the interface model of the *context-aware* meta-program, where *w* is the weight of a parameter to model the context of the parameter use. This model is also the parameter interaction model.

Definition 3.11. The graph $H((V_i, V_j), E)$ is the parameter values interaction graph.

Definition 3.12. The meta-body model is an ordered set of functions: $\mu(M_B) = \{f_k(a_j)\}$, where f_k – are constructs or functions of the meta-language and a – is the argument of a function.

In Fig. 3.3, is the feature model related to the solution domain.

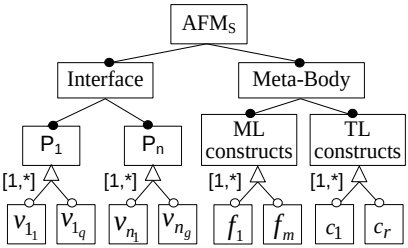

Fig. 3.3 Solution domain feature models

3.3. Transformation rules

Rule 3.1. *Variant point* in the FM_P corresponds (is equal) to a *parameter name* in the FM_S.

Rule 3.2. Variants of a variant point within the FM_P correspond (is equal) to parameter values in the FM_S.

Rule 3.3. The format of a *simple assignment statement* within the interface is as follows: <*parameter*>=<*parameter_value_set*>.

***Rule* 3.4.** The format of a *conditional assignment statement* within the interface is as follows: <*parameter1*><*condition*><*parameter2*><*parameter1*> = <*parameter_value_set*>; the conditional assignment statement appears if and only if the adequate variant point has constraints *requires* or *excludes*.

Rule 3.5. The number of parameters extracted from the model *FM*s is equal to the number of variation points extracted from the FM_P to be transferred to the engine to form the interface according to *Rule 3.3* or *Rule 3.4*.

Rule 3.6. *Abstract State Machine* (ASM) engine orders parameters (identified by *Rules 3.1, 3.2* and 3.5) according to their priorities (note that the priority feature is represented as a parameter weight, but not as the parameter itself, see *Rule 3.1*).

Rule 3.7. ASM engine presents the values of the priority parameters as a comment (/*...*/) before each simple assignment statement.

Rule 3.8. ASM engine builds the meta-program interface according to *Rules 3.1 - 3.7*.

Rule 3.9. To form the meta-body, the following set of functions of the meta-language is used: {Operation (assignment ("="), OPEN-WRITE-CLOSE), conditional, loops}.

Rule 3.10. Target language generic instance ($TLGI_{P,}$ if any) should always be written by the designer with the clear specification of the location where parameters have to appear.

Rule 3.11. In the case of $TLGI_P$ presence, the ASM engine performs parsing, i.e. syntactic analysis of the item and builds the meta-body automatically.

Rule 3.12. If there is no $TLGI_P$, ASM provides the meta-body template for its filling in by the user.

4. META-PROGRAMS SPECIALIZATION AND CONTEXTUAL ADAPTATION

Program specialization (or partial evaluation) is the technique that makes it possible to automatically transform a program into a specialized version, according to the context of use (Le Meur, Lawall and Consel, 2002).

Program specialization also relates to *stage programming* (Inoue, Taha, 2012) and *meta-programming*, especially in logic programming research. Shortly, it can be summarized as *multi-stage* programming, i.e. the development of programs in several different stages.

Futamura (1999), for example, formulates specialization task as a transformation process π as follows:

$$\pi(c_1^{'}, c_2^{'},\ldots,c_m^{'}, r_1^{'}, r_2^{'},\ldots,r_n^{'}) = \alpha(\pi, c_1^{'}, c_2^{'},\ldots,c_m^{'})(r_1^{'}, r_2^{'},\ldots,r_n^{'}) \qquad (4.1)$$

The left side of Eq. (4.1) presents the state of a program to be evaluated before specialization. Here the values $(c_1^{'}, c_2^{'},\ldots,c_m^{'}, r_1^{'}, r_2^{'},\ldots,r_n^{'})$ of variables $(c_1, c_2,\ldots c_m, r_1, r_2,\ldots r_n)$ of the program are split into two subsets: the *constants* as *compile time values* and *variables* as *run time values*. The right side of the equation specifies the state of the program after specialization using the "specialization algorithm α, which evaluates $(c_1^{'}, c_2^{'},\ldots,c_m^{'})$ in the first stage and then evaluates $(r_1^{'}, r_2^{'},\ldots,r_n^{'})$ in the second stage, though the stages are not defined explicitly. In fact, the specializer is a meta-program because it *generates* through the process π the other, i.e. a specialized program.

Now are able to formulate the meta-program specialization problem similarly. Let be given a set of parameters $P = \{(p_1,\ldots,p_m),(p_{m+1},\ldots,p_n)\}$ of a meta-program, where the space *P* is decomposed into two subsets under the following *constraint* (the subsets are not intersecting). Similarly to (4.1), it is possible to formulate the problem as *the two-stage specialization* task as follows:

$$\pi(p_1,\ldots,p_m, p_{m+1},\ldots,p_n) = \alpha(\pi, p_1,\ldots,p_m)(p_{m+1},\ldots,p_n) \qquad (4.2)$$

here, parameters $(p_1,...,p_m)$ are evaluated in *stage 2*, thus being treated as *constants*, while the remaining parameters $(p_{m+1},...,p_n)$ at this stage are treated as *variables*. To be evaluated in stage 2, parameters $(p_1,...,p_m)$ have to be *active* (meaning their usual role in the meta-program), while the remaining parameters have to be *passive* (meaning not being evaluated).

It is the role of a specializer (formally denoted as α), among others, to *pre-program* the change of states so that parameters $(p_{m+1},...,p_n)$ would be passive at *stage 2* (which describes evaluation of $(p_1,...,p_m)$ only) and they would *be active* at *stage 1* (which describes evaluation of $(p_{m+1},...,p_n)$) .

The equation (4.2) can be generalized by introducing the concept of multi-stage (e.g. *k*-stage) specialization. Therefore, it could be thought of in terms of recursion, i.e. to apply "specialization" by partitioning the remaining parameters $(p_{m+1},...,p_n)$ in two subsets (under the stated constraints) again and again until some of the *remaining parameters* will be evaluated (*k* -1) times. Consequently, the following can be written:

$$\begin{aligned}\pi(p_1,...,p_m, p_{m+1},...,p_n) = \alpha(\pi, p_1,...,p_m)(p_{m+1},...,p_n)\\ \alpha(\pi, p_{m+1},..,p_i)(p_{i+1},..,p_n)\ \dots\\ \dots\ \alpha(\pi, p_{i+1},...,p_j)(p_{j+1}...,p_n)\ \dots\end{aligned} \qquad (4.3)$$

Eqs. (4.2 and 4.3), in fact, describe the specialization not a meta-program itself but its model expressed as a parameter set. With respect to specialization through staging, however, the parameters of different type should be evaluated differently.

4.1. Fundamentals of multi-stage meta-programs

The multi-stage meta-program means constructing a meta-generator that generates the lower-level meta-programs. Here, the stage refers to as an abstraction to re-arrange the structure of a meta-program so to enable its *specialization.*

Definition 4.1. Formally, the multi-stage meta-program's structural model μ^k is a composition of the meta-interface model $\mu(M_I^k)$ and the meta-body model $\mu(M_B^k)$ (Fig. 4.1), which consists of a lower-level meta-interface models and meta-body models. Formally, it is expressed as:

$$\mu(M_B^k) = \mu(M_I^{k-1}) + (...+(\mu(M_I^1 + \mu(M_B^1)))) \qquad (4.4)$$

here *k* - number of the meta-program stages.

To specify the functional model in designing meta-meta-programs, it is needed to introduce some technological terms such as *deactivating label*, *deactivating index*, *active / passive meta-construct*.

Definition 4.2. Meta-construct is a meta-parameter or meta-function of a meta-language within the meta-body.

Definition 4.3. The label '\' or \\\ sequence ... is called deactivation label.

Definition 4.4. Meta-construct is *active* if it does not have a deactivation label, i.e. perform their defined function.

Definition 4.5. Meta-construct is *passive* if it contains the deactivating label (labels) written before the meta-construct, and do not perform their defined functions and therefore regarded as a target language text.

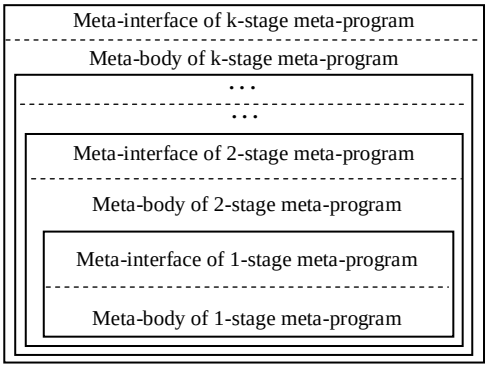

Fig.4.1 Structural model of *k*-stage meta-program

Definition 4.6. *Deactivating index* is the adequate number of deactivating labels written before a meta-construct. Formally, it is expressed as:

$$DI = 0 \text{ for stage } k;\ DI = 1 \text{ for stage } (k\text{-}1), \text{ etc. } DI = \sum_{a=0}^{k-2} 2^a, \qquad (4.5)$$

here *DI–deactivating index*.

Definition 4.7. *Deactivating process* is the multi-stage process (in terms of *k*-stage processing) to reducing the deactivating index by 1, or changing the state of a meta-function from the passive state to the active state.

Definition 4.8. Transformations $(M \xrightarrow{T} M^k)$ $(1 < k \le k_{\max})$ exist **iff** the dependency graph $G(P,U)$ of meta-program is disconnected.

Property 4.1. The upper bound of the eligible number of stages $k_{\max}$ to perform specialization of the *given correct* meta-program specification into its *k*-stage format is defined by inequality:

$$k_{\max} \le g, \qquad (4.6)$$

here g – the number of connected sub-graphs including the null graphs.

Definition 4.8. Context-based specialization is the process governed by the contextual information to define which stage is to be selected and to specify the parameter permutation to stages using the prescribed transformation rules

4.2. Rules to perform specialization transformation

Rule 4.1. The parameters and their context information are extracted from the context-aware interface model $G(P^w, U)$, $w \in \{HP, IP, LP\}$. The information is to be represented in a separate file.

Rule 4.2. Meta-parameter and contextual information data file must be created with appropriate structures.

Rule 4.3. The check expression defined in clause (4.6). If $k \le g$, then the transformation may occur, otherwise it is impossible to transform.

Rule 4.4. Dependent meta-parameter must always be assigned the same stage.

Rule 4.5. A stage is not empty, i.e. it has at least one parameter group or a separate parameter.

Rule 4.6. The group of parameters with the highest priority (HP) should appear at the *higher stages.*

Rule 4.7. The group of parameters with the intermediate priority (IP) or with the lower priority (LP) should appear at the *lower stages.*

Rule 4.8. The number of stages and the parameters' group allocation to stages are performed automatically according to the context information (i.e. according to the parameter priorities).

Rule 4.9. The number of stages and the allocation of the parameters to stages can also be performed by the user.

Rule 4.10. *Rule 4.8* and *Rule 4.9* are mutually exclusive

Rule 4.11. When the parameter allocating process runs at stage *i*, all parameters are to be deactivated by the deactivating index at stages (*i*-1) … 1.

Rule 4.12. Each deactivated parameter *requires* the deactivating of the meta-function (within the meta-body) with the same deactivating index, in which this parameter appears.

4.3. Meta-program adaptation

The aim of this transformation is to make possible the pre-programmed user-guided adaptation of meta-programs when used. The specialization process results in creating the multi-stage executable specification that is coded as the *k*-stage heterogeneous meta-program. Specialization of meta-program by staging enables to automatically prepare the content for the different contexts of use.

Content adaptation is the user-guided process that includes user's actions and automatic processing by the tool. The user views the given interface of meta-program so that to recognize and supply his / her context parameter values. Then the automatic processing follows; yielding more *specialized variants* to support the needs for adaptation (Fig.4.2).

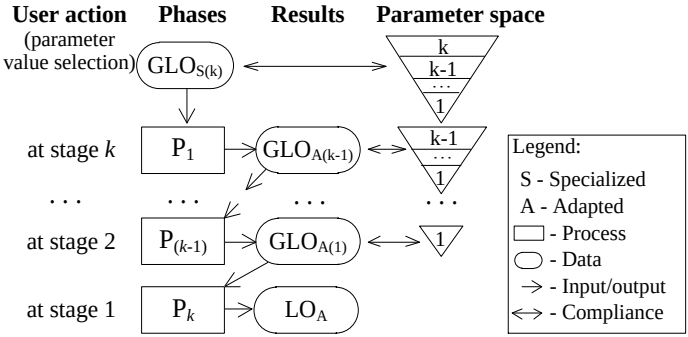

Fig. 4.2 Stage-based adaptation processes

5. TRANSFORMATION TOOLS: CREATION AND EVALUATION

5.1. Meta-program development tool „MePAG“

The tool „MePAG“ (**Me**ta-**P**rogram **A**utomatic **G**enerator) supports the transformation M2MP (meaning *model*-to-*meta-program*, i.e. transformation lowering of the abstraction level). The transformation process is semi-formal because not all input data used it is difficult, or even impossible, to present formally. The reason is that the heterogeneous meta-program generation paradigm used, in which the meta-language and the target language, are both abstract (not formal). Furthermore, not always is possible to synthesize a meta-program fully automatically.

A standard meta-language processor (e.g. PHP-processor in our case, though other languages such as C++, Java) can be used in the role of a meta-language serves as a generating tool to provide the experimental validation of a synthesized meta-program. This process may be multi-cycle with feedback possible. This may happen due to some semantic or syntactic inconsistency introduced by the designer when such an interleaving is needed.

The technique enables the development of a higher-level executable specification (i.e. meta-program) from which target program instances are generated on demand automatically, at the use phase.

The tool „MePAG“ that supports the approach enables it to synthesize heterogeneous meta-programs from two *input feature models* and supplementary data, such as constructs of the meta-language and target language (in generic instance model TLGI). One feature model, namely FM_P, and $TLGI_P$ represent the *problem domain*; whereas the other feature model, namely FM_S, and meta-languagefunctions MLF_S (see Fig. 5.1) represent the *solution domain*.

Two additional properties of the input models are important to state: 1) it is possible to create $TLGI_P$ easily (for not complex tasks), 2) there are difficulties

in creating $TLGI_P$ for real world tasks when the efforts of developing the model $TLGI_P$ are roughly the same as manual coding of the meta-program meta-body.

Typically, the first case means that it is able to develop the relatively simple meta-programs automatically. The second case is more general and specifies the real-world tasks for which we are not always able to develop meta-programs automatically, or such a mode is merely unreasonable due to the complexity issues as defined previously.

There is some difference among the models FM_P and FM_S in terms of their mode of use. The first model is created *anew* for each new problem task to be solved; whereas the second model is common for all domain tasks considered in the given context. Therefore, due to this property, it is able to represent the model FM_S within the transformation engine as a *fixed data structure* while the model FM_P should be always be supplied to the engine as the external input model (Fig. 5.1).

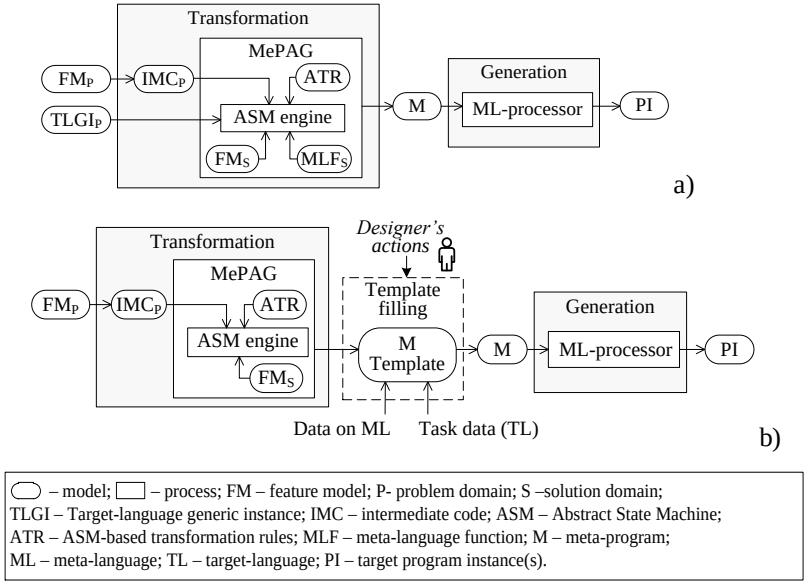

Fig. 5.1 „MePAG" tool working modes: a) automatic, b) semi-automatic

5.2. Functioning algorithm of the tool „MePAG"

Step 1. **if** <IMC_P exists> **then** Read data; /* *Rule 3.5*; *value of n* ($n \geq 1$) *is defined* */
Step 2. **if** n >1 **then** Sort parameters according to their priorities; /**Rule 3.6* */
Step 3. Create the MP file /**MP.php*/ to store MP's statements;
Step 4. Write a comment to denote the beginning of the interface; /* *for template filling in*/
Step 5. **for** *i* = 1 to *n* **do**

Read data for the parameter *i;* /**Rule 3.1 & Rule 3.2**/
if <parameter independence exists> **then** Create the parameter value selection form; /* *Rule 3.3* & *Rule 3.7**/
else Create the parameter value selection form with the conditional branching; /* *Rule 3.4* & *Rule 3.7**/
end;

Step 6. Create a comment to denote the beginning of the meta-body; /* *for template filling in**/

Step 7. **if** <$TLGI_P$ not exists> **then do** Create comments for the user; Create the MP completion statements; **end do**; /**Rule 3.12* */
else do Read the $TLGI_P$ and make the parser's initialization; /**Rule 3.10*; *value of m* (*m* >1) *is defined* */
for *i* = 1 to *m* **do**
Perform parsing the line *i* within $TLGI_P$;
Find the parameter locations in the line and create parameter variables; /**Rule 3.11**/
Create the meta-body line; /**Rule 3.9**/
end;
Create the MP completion statements;
end do;

Step 8. **end.**

5.3. Meta-program specialization tool „MP-ReTool"

On the basis of the specialization process and the theoretical background, an experimental tool „MP-ReTool" (Fig. 5.2) has been developed that transforms a meta-program into its *k*-stage representation as follows: 1→2, 1→3 , 1→4 and 1→5. Here, the numbers define stages. The tool enables the saving of a great deal of the user's efforts and resources because: 1) the manual process is error-prone and time-consuming and 2) the direct manual refactoring (e.g. 1→4) is practically unachievable due to readability issues.

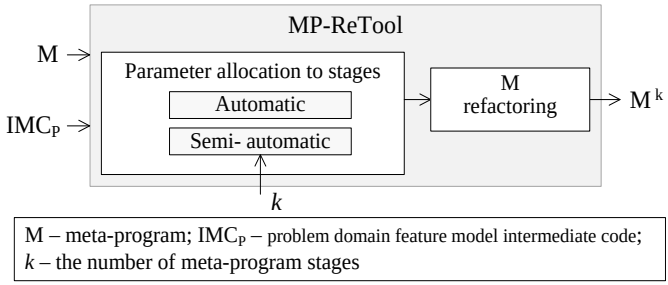

Fig. 5.2 Structure of „MP-ReTool"

The tool implements the user-tool communication model to solve the specialization problem. There are two modes of using the tool. In mode 1, the user (typically teacher) indicates (through the communication model) how the

meta-parameters are to be allocated to stages. In mode 2, the tool works fully automatically. In this case, however, parameters are to be supplied *with non-redundant weights* introduced by the meta-program designer when the specification is coded.

5.4. Algorithm to perform refactoring transformation

*Step 1***.** *Choose* the operating mode; /* *mode 1* or *mode 2**/
*Step 2***.** *Read* the (meta-)interface model; /* *Rule 4.1, Rule 4.2; the model is created by* „MePAG“*/
*Step 3***.** *Read* the meta-body;
Step 4. **if** <*mode 1*> **then do** *Initiate* parameters' assignments to stages; /**Rule 4.8* */
if n >1 **then** *Sort* parameters according to their priorities; /**n* – the number of parameters*/
Identify the number of required stages; /*according to the priority values from the set: {HP; IP (L1); IP (L2); IP (L3); LP (L4, L5, L6)} */
Allocate parameters to stages; /* according to the parameter priority values; *Rule 4.4 - 4.7**/
if <there is no refactoring feasibility> **then** *Print* error message 'correct the model, i.e. change the priority values and start from the beginning'
end;
Step 5. **if** <*mode 2*> **then do** *Initiate* parameters' assignments to stages; /**Rule 4.9* */
Choose the number *k*; /* *k* is # of required stages */
if *k* >*g* **then** *Print* message 'refactoring impossible: *reduce k* ';
/* *Rule 4.3*; *g* – the maximum number of stages*/
if n >1 **then** *Sort* parameters according to their priorities;
Allocate parameters to stages; */ according the user's choice; the priority values from the set: {HP; HP or IP; IP; IP or LP; LP}; *Rules 4.4 - 4.7* */
if < allocation is incorrect> **then** *Print* error message: allocate parameters anew*;*
end*;*
*Step 6***.** *Perform* the meta-interface refactoring as follows;
for *i* = 1 to *n* **do**
Read data of the parameter *i;*
Fix the parameter to the given stage (which will be used);
if <parameter *i* is independent > **then** *Create* the *simple* interface form for this stage; /* the parameter *i* deactivation; *Rule 4.11* */
else *Create* the *branching* interface form for this stage; /* the parameter *i* deactivation; *Rule 4.11* */
end;
*Step 7***.** *Perform* the meta-body refactoring as follows;
for *j* = 1 to *m* **do** (*m* – the number of meta-body code lines)
Perform parsing of the meta-body line *j*;
if <any parameter in the line *j* exists> **then**
Fix the parameter stage from the *staged interface;* /* it has already been formed at *Step 6* */
if <the parameter stage is less (<) than *k*> **then**
Deactivate the parameter and its all functions; /* *Rule 4.12**/

else *Rewrite* the line *j* without changes;
else; *Rewrite* the line *j* without changes;
end;
Step 8. **end.**

6. EXPERIMENTAL EVALUATION

This section presents *a methodology* of the experiments and results obtained applying the manual development and using the developed tool. Experiments were carried out with real tasks to investigate both the meta-programs and the tool. The experience of the author (authors) was about 5 years in robot-based programming and meta-programming. Table 6.1 presents the comparison of meta-program design modes.

Table 6.1. Attribute-based comparison of meta-program design modes

M design mode / **Attributes**	**Manual 1**	**Manual 2**	**Semi-automatic**	**Automatic**
Input data	TL, ML, R, C	TL, ML, R, C	TL, ML, R, C	TL, ML, R, C
Input data representation	Explicit by example	Explicit by scenario	Explicit by FMs	Explicit by EFMs
Models constructing	Ad hoc based on intuition & Designer's knowledge	Systemized based on models and manual design rules	Systemized based on models and Tool (Me-PAG)	Systemized based on extended models and Tool (Me-PAG)
Models input data	Implicit in designer's mind	Explicit problem domain FM	D*f*R and D*w*R framework, Learning variability FMs, CBFM	The same as previous plus Solution domain models, CBFM
Transformation rules	Intuitive rules based on designers competence, implicit context	Explicit rules, explicit context	Tool supported basic set of transformation rules, context FM	Tool supported extended set of model transformation Rules, context FM
Transformation engine	No computational model	Week human –oriented computational model	ASM-based computational model comprising the interface design only	ASM-based computational model comprising the interface and meta-body design

Legend: M – meta-program; TL– target language; ML– meta-language, Rs – requirements, Cs – constraints; FM – feature model; CBFM – context-based feature model; ASM- abstract state machine

The aim was twofold: to test correctness of meta-programs and to test the correct functionality of the tool „MePAG" through the solving of real world tasks.

Teaching and learning in CS basic courses were selected as a problem domain, using meta-programs as higher-level Learning Objects to generate the

Learning Objects instances on demand. To support advanced learning in CS, LEGO NXT-based and ARDUINO-based educational environments were used. The task specification (i.e. feature models) was developed by the domain experts.

In Fig. 6.1, the needed efforts to design meta-programs of three modes (manual design, semi-automatic and automatic design) for the same version for all selected tasks are presented. The average efforts expressed by the time dimension are provided.

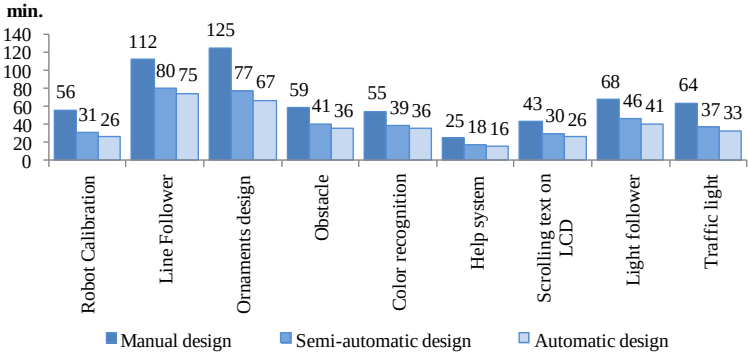

Fig. 6.1 The needed efforts to design meta-programs three modes

The obtained comparative evaluation of manual design with semi-automatic design and manual design with automatic design are at Fig 6.2. The semi-automatic development of SLOs is more efficient by 30 – 46 % as compared to the pure manual development. The automatic development is more efficient by 33 – 54 % as compared to the automatic development. The automatic mode gains compared with the semi-automatic mode is evaluated by 6 – 13 %.

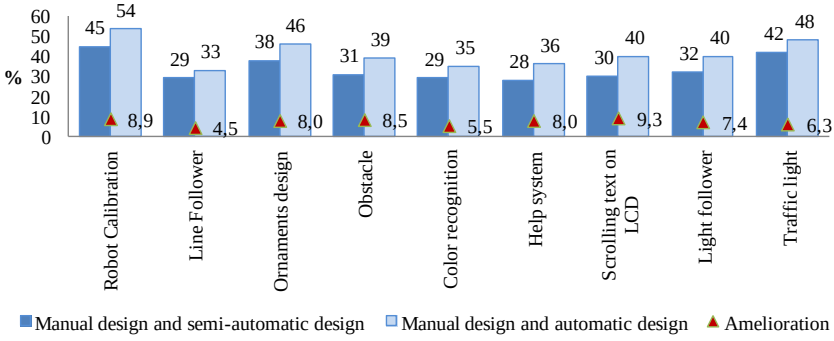

Fig. 6.2 Comparative evaluation of design modes

CONCLUSIONS

1. Analysis of the related work has shown that:
   - It was identified that the essential requirement to deal with the problem domain tasks in designing systems is the identification and specification of commonality-variability relationships in representing models for transformation.
   - As meta-programming enables the achievement of a high-level automation in creating programs (meta-program is a program generator), the following problems (meta-programs creation through model-based transformations and meta-program specialization through adaptation-based transformation) stand for research tasks of the dissertation.
2. The task of creating heterogeneous meta-programs has been formulated as the *multistage transformation* of two model types. The first model represents the problem domain, whereas the second represents the solution domain, i.e. meta-programming. For each domain, the created feature models and the following theoretical result has been achieved: the formal description of the models with the identified properties and relationships. The latter enables to formulate the transformation rules to create the adequate algorithms and tools.
3. The meta-program specialization task has been formulated on the basis of the *Futumura* program specialization task. Furthermore, by applying the ideas of multistage programming, it was possible to formulate the meta-program specialization task for the general case, i.e. as a multi-stage transformation task. This generalization is treated as a new scientific result, because so far there has only been known the two-stage meta-program transformation task.
4. The essential theoretical result of the multistage meta-program transformation (specialization) task is formulated as follows:
   - The task *solvability condition* is: the solution exists if and only if the weighted graph $G(P^w, U)$, representing the meta-interface of the original meta-program, is a disconnected graph (here P – set of meta-parameters, U – set of edges that represents the interaction among meta-parameters, w – a variable of fuzzy logic describing the meta-parameter context of use).
   - It was identified that the maximum number of stages is equal to the total number of the components of the graph $G(P^w, U)$.
   - To solve the problem, the principle of deactivation-activation of meta-constructs and identification of the value of the deactivating index (*DI*) were applied:

For the stage k, $DI = 0$, for the stage (k-1), $DI = 1$, and for the remaining stages $DI = \sum_{a=0}^{k-2} 2^a$.

5. The suggested, tested and applied tools („FAMILIAR“ and „SPLOT“ – have been selected, „MePAG“ and „MP-ReTool“ have been created) support the complete meta-program life cycle: modeling, model transformation into meta-program, meta-program transformation, generation and maintenance.
6. It was identified that using the tool „MePAG“ obtained an efficiency increase in creating meta-programs by 34 % on average as compared to the manual process.
7. The conducted experiments with the meta-program transformations into the multistage representation have proved the hypothesis that the meta-program specialization changes only its structure, preserving its initial functionality.
8. The tool „MP-ReTool“ is for automatic transformation of one-stage meta-program into the multistage one. Such a kind of transformation enables to automatically create the meta-programming-based meta-generators and investigate meta-program adaptation problems for the use context.
9. The investigation on the complexity evaluation and complexity changes as related to the introduced methods using the known complexity measures has been provided. It was identified that with the increase of stages – complexity is increasing too (Cognitive Difficulty in higher stage increases more than 50 %); however, the understandability is diminishing significantly and the only way to deal with the problem is the use of the developed tool. Also some difficulties of the investigated approach (e.g., the initial model discovery, etc.) have been identified for the future work.

REFERENCES

1. ACHER, M., COLLET, P., LAHIRE, P. and FRANCE, R. B. (2013). Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming*, 78.6: 657-681.
2. BIEHL, M. (2010). Literature study on model transformations. *Royal Institute of Technology, Tech. Rep. ISRN/KTH/MMK*.
3. CAPILLA, R., BOSCH, J., KANG, K. C. (2013). *Systems and Software Variability Management*. Springer.
4. FIORAVANTI, F., PETTOROSSI, A., PROIETTI, M. and SENNI, V. (2011). Program transformation for development, verification, and synthesis of programs. *Intelligenza Artificiale*, 5.1: 119-125.
5. FUTAMURA, Y. (1999) Partial evaluation of computation process--an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12.4: 381-391.

6. HARMAN, M. (2010). Why Source Code Analysis and Manipulation Will Always be Important. In: *Source Code Analysis and Manipulation, SCAM*. p. 7-19.
7. INOUE, J., TAHA, W. (2012). Reasoning about multi-stage programs. In: *Programming Languages and Systems*. Springer Berlin Heidelberg, p. 357-376.
8. LE MEUR, A. F., LAWALL, J. L., CONSEL, C. (2002).Towards bridging the gap between programming languages and partial evaluation. In: *ACM SIGPLAN Notices*. ACM, p. 9-18.
9. POHL, K., BÖCKLE, G., VAN DER LINDEN, F. J. (2005). *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.
10. SAMETINGER, J. (1997). *Software engineering with reusable components*. Springer Science & Business Media.
11. ŠTUIKYS, V., DAMAŠEVIČIUS, R. (2013). *Meta-Programming and Model-Driven Meta-Program Development: Principles, Processes and Techniques*. Springer Science & Business Media.
12. VISSER, E. (2001). A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57: 109-143.
13. VÖLTER, M., STAHL, T., BETTIN, J., HAASE, A. and HELSEN, S. (2013). *Model-driven software development: technology, engineering, management*. John Wiley & Sons.
14. ZHANG, X. (2014). Developing Model-Driven Software Product Lines. PhD. University of Oslo.

LIST OF PUBLICATIONS ON THE SUBJECT OF DISSERTATION

Publications in journals included into the Institute for Scientific Information (ISI) database

1. Štuikys, V., Bespalova, K., & Burbaitė, R. (2014). Refactoring of Heterogeneous Meta-Program into k-stage Meta-Program.*Information Technology and Control*. ISSN 1392-124X. *43*(1), p. 14-27. [ISI Web of Science; INSPEC].

2. Burbaitė, R., Bespalova, K., Damaševičius, R., & Štuikys, V. (2014) (n.d.). Context-Aware Generative Learning Objects for Teaching Computer Science. Accepted to*International Journal of Engineering Education*. *30*(4), p. 929–936. [ISI Web of Science; Scopus].

Articles referred in other international databases

1. Burbaite, R., Damasevicius, R., Stuikys, V., Bespalova, K., & Paskevicius, P. (2011). Product variation sequence modelling using feature diagrams and modal logic. CINTI 2011: 12th IEEE International Symposium on

Computational Intelligence and Informatics, November 21-22, 2011, Budapest, Hungary: proceedings. Budapest: IEEE, 2011. ISBN 9781457700439. p. 73-77. [IEEE/IEE].

2. Štuikys, V., & Bespalova, K. (2012). Methodology and Experiments to Transform Heterogeneous Meta-program into Meta-meta-programs. Information and software technologies: 18th International Conference, ICIST 2012, Kaunas, Lithuania, September 13-14, 2012: proceedings / [edited by] Tomas Skersys, Rimantas Butleris, Rita Butkiene. Berlin, Heidelberg: Springer, ISBN 9783642333071. p. 210-225. [Conference Proceedings Citation Index]

3. Štuikys, V., Bespalova, K., & Burbaitė, R. (2014). Generative Learning Object (GLO) Specialization: Teacher's and Learner's View. Information and software technologies: 20th International Conference, ICIST 2014, Druskininkai, Lithuania, October 9-10, 2014: proceedings / [edited by] Giedrė Drėgvaitė, Robertas damaševičius, Springer International Publishing, 2014. ISBN 9783319119571. p. 291-301. [Conference Proceedings Citation Index]

4. Burbaitė, R. & Bespalova, K.(2014). Model-Driven Processes and Tools to Design GLO for CS Education. SIIE 2014: XVI International Symposium on Computers in Education, November 12-14, 2014, Logrono, La Rioja, Spain: proceedings. Logrono: IEEE, 2014. p. 193-199. [IEEE/IEE]

5. Štuikys, V., Bespalova, K., & Burbaitė, R. (2014). Feature Transformation-Based Computational Model and Tools for Heterogeneous Meta-Program Design. CINTI 2014: 15th IEEE International Symposium On Computational Intelligence and Informatics, November 19-21, 2014, Budapest, Hungary: proceedings. Budapest: IEEE, 2014. p.185-190. [IEEE/IEE]

6. Bespalova, K., Štuikys, V. & Burbaitė, R. (2015). CS-Oriented Robot-Based GLOs Adaptation through the Content Specialization and Generation.IFIP TC3 Working Conference A New Culture of Learning: Computing and Next Generations, July 1-3, 2015, Vilnius, Lithuania

INFORMATION ABOUT THE AUTHOR OF THE DISSERTATION

Education:

2002 – 2007: gained Bachelor in informatics engineering at Kaunas University of Technology.
2007 – 2009: gained master degree in informatics engineering at Kaunas University of Technology.
2011 – 2015: doctoral studies in informatics at Kaunas University of Technology.

Work experience:

1999 – 2004: engineer, Kauno kolegija / University of Applied Sciences.
2004 – 2013: methodologist, Kauno kolegija / University of Applied Sciences.
2013 – 2014: head, unit of information system management, Kauno kolegija / University of Applied Sciences.
2014 – 2015: head, unit of study programs, Kaunas University of Technology.
Since 2015: head, unit of study administration, Kauno kolegija / University of Applied Sciences.

E-mail: kristina.bespalova@ktu.edu

REZIUMĖ

Darbo aktualumas

Transformavimas vienos formos objektų (procesų, energijos, gaminių ir pan.) į kitą yra esminis visų techninių sistemų atributas. Informatikoje šis atributas dar svarbesnis dėl šių priežasčių: (1) transformavimo objektai yra ne fiziniai objektai, o jų abstraktūs atvaizdavimai (duomenys, programos, modeliai); (2) egzistuoja gausybė atvaizdavimo formų; (3) abstraktūs atvaizdavimai įgalina daug lengviau įgyvendinti transformavimus; (4) transformavimas informatikoje lemia praktiškai visų kompiuterinių sistemų funkcionalumą.

Transformavimas informatikoje naudojamas įvairiuose kontekstuose ir apima labai platų spektrą, nuo žemiausio iki aukščiausio lygmens. Žemiausiame lygmenyje yra tradicinės transformacijos: procesoriaus, operacinės sistemos. Aukštesniame lygmenyje yra kompiliavimo transformacijos, dar aukščiau – taikomųjų sistemų projektavimo transformacijos, o aukščiausiame – sistemų-sistemų lygmens transformacijos. Programų inžinerijoje ir informatikoje esminės transformacijos atliekamos su programomis ir modeliais. Programos transformavimas yra taikomas konstruojant, optimizuojant, programų sintezėje, pertvarkyme, programinės įrangos atnaujinime, apgrąžos inžinerijoje ir kt. (Visser, 2001).

Transformavimo tyrimai labai platūs, tačiau galima teigti, kad jų visų tikslas vienas – padidinti kuriamų sistemų našumą ir efektyvumą. Pagrindinis transformavimo siekis – automatizavimas.

Per pastarąjį dešimtmetį mokslo ir technikos srityje stebimas ryškus informacinių technologijų šuolis. Bazinių technologijų raida (turima omenyje lustus) pranoko visus lūkesčius. Šiandien mes jau naudojamės tuo pagrindu sukurtomis naujomis technologijomis, gyvename ir dirbame skaitmeniniame pasaulyje, kuriame pokyčiai yra pastovus reiškinys. Vystantis informacinėms technologijoms (IT) sparčiai auga IT vartotojų kategorijos, atsiranda vis didesnis poreikis kuriamas sistemas pritaikyti prie rinkos reikalavimų. Dar viena esminė ypatybė – nepaliaujamai auga programinio kodo svoris (apimtis) sistemose. Tai geriausiai matoma įterptinėse sistemose (pvz., realaus laiko) ir internetiniuose taikymuose (pvz., daiktų internetas).

Sparti technologijų raida ir rinka taip pat lemia projektuojamų sistemų sudėtingumo, dydžio ir sąveikos laipsnio bei programinio kodo augimą. Tai yra dideli iššūkiai sistemų kūrėjams. Koks galimas atsakas į šiuos iššūkius? Technologijų raidos patikrintas atsakas – abstrakcijos lygmens kėlimas tiek nagrinėjant probleminę sritį (t. y. taikymus), tiek sprendimų sritį (t. y. metodus). Todėl kuriamas sistemas siekiama atvaizduoti aukštesniu abstrakcijos lygmeniu, kuriami nauji projektavimo metodai, sistemos sudalijamos į atskiras dalis (koncepcijų atskirtis), kuriant naudojami automatiniai transformavimo įrankiai. Aukštesnis abstrakcijos lygmuo įvairiuose kontekstuose mokslinėje literatūroje

įvardijamas kaip metalygmuo (pvz., plačiai naudojamos sąvokos metamodelis, metaduomenys, metakalba, metaprograma ir kt.).

Kita vertus, šiuolaikinių sistemų kūrimas grindžiamas pakartotinio naudojimo (angl. *reuse*) metodologija. Ši metodologija remiasi programų šeimynos koncepcija (jas galima traktuoti kaip metasistemas), apima srities analizę, modeliavimą (modelių ir metamodelių sukūrimą), bendrųjų (meta) komponentų bei programų generatorių kūrimą.

Pastaraisiais metais dominuoja dvi pakartotiniu naudojimu grindžiamos kūrimo metodologijos: OMG modelių inžinerija (angl. *Model-Driven Engineering*, MDE), kuri paprastai remiasi objektinėmis abstrakcijomis (UML standartas) ir požymių modelių inžinerija, kuri labiau išryškina ir akcentuoja programų šeimynų koncepciją (angl. *Product Line Engineering*, PLE) (Pohl ir kt., 2005). Abi metodologijos nagrinėjamos dviejuose lygmenyse (srities inžinerijos ir taikymų inžinerijos) skatina visame kūrimo procese naudoti aukšto lygmens modelius, jų transformavimą užtikrinant sisteminį pakartotinį panaudojimą, t. y. siekiant aukštesnio automatizavimo laipsnio, didesnio našumo ir kokybės. Nustatyta, kad modeliais grįstos metodologijos yra vyraujančios naujausiuose tyrimuose. Čia dar daug neišspręstų problemų, siejamų su analize, variantiškumo modeliavimu (Capilla, Bosch ir Kang, 2013), atvaizdavimu, transformavimu ir realizacija (Biehl, 2010; Fioravanti ir kt. 2011; Völter ir kt., 2013; Zhang, 2014).

Disertacijoje keliami ir nagrinėjami uždaviniai yra specifiniai, mažai tyrinėti šių metodologijų atvejai: požymių modelių transformavimas į heterogenines metaprogramas, vidinis metaprogramų transformavimas siekiant jų adaptavimo prie konkretaus taikymo. Metaprogramų kūrimo ir tobulinimo procesai sudėtingi, reikalauja gilių žinių tiek iš taikomosios, tiek iš sprendimo srities. Dėl to metaprogramų kūrimo, transformavimo ir palaikymo procesus tikslinga automatizuoti. Kita vertus, šio tipo metaprogramų automatizuotas kūrimas remiantis požymių modelių transformavimu, mūsų žiniomis, išvis nebuvo nagrinėtas. Kadangi metaprogramos yra tikslo (srities) programų generatoriai, todėl galima drąsiai tvirtinti, kad disertacijoje pasirinkta tema yra aktuali ir savalaikė.

Tyrimo objektas

Darbe tiriama probleminės srities požymių modeliai, metaprogramos, jų kūrimo ir transformavimo procesai ir metodai.

Darbo tikslas

Darbo tikslas – sukurti ir ištirti heterogeninių metaprogramų automatizuoto kūrimo ir transformavimo metodiką, įskaitant tuos procesus palaikančius įrankius

Darbo uždaviniai

1. Išanalizuoti ir įvertinti modelių ir programų (metaprogramų) transformavimo metodus.
2. Sukurti ir ištirti heterogeninių metaprogramų kūrimo metodą panaudojant požymių modelių transformacijas.
3. Sukurti ir ištirti metodą, kuris transformuotų vienpakopę heterogeninę metaprogramą į daugiapakopę.
4. Sukurti ir ištirti metaprogramų kūrimo ir transformavimo algoritmus ir juos realizuoti atitinkamuose įrankiuose.

Ginamieji teiginiai

1. Probleminės ir sprendimo sričių požymiais grindžiami modeliai įgalina metodiškai kurti metaprogramas automatizuojant kūrimo procesą.
2. Metaprogramų specializavimo ir adaptavimo formalieji modeliai užtikrina transformavimo įrankio funkcionalumą ir korektiškumą.
3. Sukurti ir išbandyti transformavimo įrankiai užtikrina efektyvų metaprogramų kūrimą, transformavimą ir palaikymą.

Mokslinis naujumas

1. Pasiūlytas ir ištirtas heterogeninių metaprogramų automatizuotas kūrimo metodas, kuris remiasi požymių modelių transformacijomis.
2. Nustatyta daugiapakopės transformacijos uždavinio sprendinių egzistavimo sąlyga bei maksimalus leistinas pakopų skaičius, įgalinantis apibendrinti dvipakopę metaprogramų transformaciją į daugiapakopę.
3. Pasiūlytas išbaigtas procesas, apimantis (i) modelių ir metaprogramų sukūrimą (panaudojant sukūrimo įrankį), (ii) jų transformavimą (panaudojant restruktūrizavimo įrankį) ir (iii) iš metaprogramų sugeneruotų programų pritaikymą.

Praktinis naujumas

1. Automatizuotas metaprogramų kūrimas (įrankis) ir automatinis mokomųjų robotų valdymo programų generavimas.
2. Automatizuotas daugiapakopių metaprogramų kūrimas (įrankis) ir automatinis metaprogramų bei mokomųjų robotų valdymo programų adaptavimas panaudos kontekstui.

IŠVADOS

1. Atlikus literatūros šaltinių analizę nustatyta, kad:
   - esminis reikalavimas kuriamiems probleminės srities modeliams ir jų transformavimui yra *bendrybių-skirtybių ir jų sąveikos identifikavimas*, *kaip tiriamosios srities variantiškumo išraiška*;
   - heterogeninis metaprogramavimas įgalina pasiekti programų kūrimo automatizavimo tikslus, o programų generatoriai realizuoja generatyvinį pakartotinį panaudojimą.

2. Sukurti bendrybes ir skirtybes aprašantys probleminės ir sprendimų srities formalizuoti požymių modeliai, jų sąryšiai, savybės ir požymiais grindžiamų modelių transformavimo taisyklės įgalino automatizuotai kurti heterogenines metaprogramas.
3. Pritaikytas *Futamuros* programų specializavimo uždavinio *interpretavimas* įgalino suformuluoti dviejų pakopų metaprogramų specializavimo uždavinį, po to, pastarajam pritaikius apgrąžos principą, suformuluotas pradinės (vienpakopės) metaprogramos daugiapakopio transformavimo (t. y. specializavimo) uždavinys. Kadangi dviejų pakopų metaprogramos modelis (kitoje notacijoje) jau buvo žinomas, tai šis apibendrinimas yra moksliškai naujas.
4. Daugiapakopio transformavimo esminis teorinis rezultatas apibrėžiamas taip:
   - nustatyta apibendrinto specializacijos uždavinio išsprendžiamumo sąlyga, t. y. „*uždavinys išsprendžiamas tada ir tik tada, jei vienpakopės metaprogramos metasąsajos svorinis grafas* $G(P^w, U)$ *nėra jungusis grafas*“ (čia P – metaparametrų aibė, U – briaunų aibė, vaizduojanti metaparametrų sąveiką, w – neraiškiosios logikos kintamasis, aprašantis metaparametro kontekstą);
   - nustatyta, kad *maksimalus pakopų skaičius* lygus metasąsajos grafo *visuminiam komponenčių skaičiui* (t. y. jungias ir nejungias komponentes kartu paėmus);
   - uždaviniui išspręsti pritaikytas metakonstrukcijų deaktyvacijos-aktyvacijos principas ir nustatyta pakopos deaktyvacijos indekso (*DI*) reikšmė duotai metakalbai, t. y. pakopoje k $DI = 0$, pakopoje (k-1) $DI = 1$, o žemesnėse pakopose $DI = \sum_{a=0}^{k-2} 2^a$.
5. Pasiūlyti, išbandyti ir pritaikyti įrankiai (vieni – „FAMILIAR“ ir „SPLOT“ parinkti, kiti – „MePAG“ ir „MP-ReTool“ sukurti), palaikantys pilną metaprogramos gyvavimo ciklą: *modeliavimo, modelių transformavimo, metaprogramų transformavimo į daugiapakopes*.
6. Nustatyta, kad naujai sukurto įrankio „MePAG“ panaudojimas leido metaprogramas kurti efektyviau. Metaprogramas kuriant pusiau automatiniu būdu sugaištama vidutiniškai 34 % mažiau laiko nei kuriant rankiniu būdu. Pusiau automatinio ir automatinio kūrimo būdų laikai labai artimi, nes automatinis būdas reikalauja didesnių laiko sąnaudų modeliams sukurti. Nustatyta, kad tikslo kalbos bendrinį programos egzempliorių tikslinga kurti tada, kai žinome, kad metaprograma bus ne kartą kuriama pakartotinai.
7. Atliktas metaprogramos transformavimo į daugiapakopę metaprogramą ekvivalentiškumo tyrimas patvirtino hipotezę, kad metaprogramos

specializavimas keičia metaprogramos struktūrą, tačiau išsaugo pradinį metaprogramos funkcionalumą.

8. Nustatyta, kad naujai sukurtas įrankis „MP-ReTool“ leidžia automatizuotai transformuoti vienpakopę metaprogramą į daugiapakopę. Šios transformacijos dėka sukuriama specializuota metaprograma, kuri įgalina metaprogramas adaptuoti prie skirtingo konteksto.
9. Atlikus heterogeninių metaprogramų technologinio sudėtingumo tyrimą nustatyta, kad didėjant metaprogramos pakopų skaičiui didėja metaprogramos sudėtingumas. *Pažinimo sudėtingumo* metrikos vertė kiekvienoje aukštesnėje pakopoje auga daugiau nei 50 %. Tai parodo, kad metaprogramą transformuojant į viena pakopa aukštesnę metaprogramą, dvigubai mažėja jos suprantamumas.

UDK 004.4’24 (043.3)
SL344. 2015-11-04 2 leidyb. apsk.1. Tiražas 50 egz. Užsakymas 392.
Išleido Kauno technologijos universitetas, K. Donelaičio g. 73, 44249 Kaunas
Spausdino leidyklos „Technologija“ spaustuvė, Studentų g. 54, 51424 Kaunas