ISSN 1392 – 124X INFORMATION TECHNOLOGY AND CONTROL, 2012, Vol. 41, No. 1

UNDERSTANDING OF HETEROGENEOUS MULTI-STAGE META-PROGRAMS

Vytautas Štuikys, Robertas Damaševičius, Giedrius Ziberkas, Kęstutis Valinčius

Software Engineering Department,
Kaunas University of Technology (Lithuania)
e-mail: vytautas.stuikys@if.ktu.lt, robertas.damasevicius@ktu.lt, ziber@soften.ktu.lt, kestutis.valincius@gmail.com

crossref http://dx.doi.org/10.5755/j01.itc.41.1.916

Abstract. The paper analyzes an approach to understanding heterogeneous meta-programs and multi-stage meta-programs. At the core of the approach is human-centred analysis combined with the Brook's program cognition theory and the concept of reverse engineering. The use of the approach leads to extracting higher-level models (graphs representing meta-parameter — meta-function relationship models, feature diagrams and algorithms) from correct meta-specifications. The models and processes enable not only to better understand the multi-stage heterogeneous meta-programs but also to contribute to their evolution. The paper describes some properties of the multi-stage heterogeneous meta-programs. The approach is supported by the case study and complexity evaluation.

Keywords: Meta-program comprehension; multi-stage heterogeneous meta-program; reverse engineering, meta-program complexity.

1. Introduction

Program understanding (aka comprehension) is an increasingly important theme due to many reasons: 1) the number and size of software projects is growing; 2) software complexity is increasing; 3) professional software developers are estimated to spend over 75% of their time trying to understand code [1, 2]; 4) it is impossible to apply reuse-based development (which prevails in modern software development methodologies, such as Product Line Engineering (PLE) [3]) without understanding of software components, component generators, component generation and integration processes [4]; 5) disciplines such as reverse engineering [5], software evolution [6], testing, etc., in fact, are based on program comprehension; and 6) it is important for knowledge extraction and learning [7].

Program understanding, however, is a hard cognitive task. Understanding even of a relatively small program is a complex process. It requires both knowledge and analysis of the programming language syntax, semantics, the computing machine and its operating environment, the application domain, etc. Typically, even for an expert a single pass over a program is insufficient to understand it. Multiple passes can help to answer global questions concerning variable use, execution paths, etc.

Meta-program is a specific case of a program generalization [8] and can be seen as a program generator [9]. Meta-programs generating other programs are much more complex than the items they produce. Therefore, the understandability of meta-programs is as important as comprehension of a program (or even at a larger degree knowing the fact that the extent of using generators is further expanding [4, 10]).

Program (meta-program) understandability and complexity are related: e.g., when complexity grows, understandability tends to diminish. As for the intention to use meta-programs, one important reason should be stated in this context (e.g., user's viewpoint): meta-programming helps the effective managing of complexity through the automatic generation of program instances. On the other hand, the complexity of a meta-program *per se* tends to grow as a logical consequence of the increase of the complexity of systems. As complexity of product families grows (in terms of features and their variants, especially due to the need to combine target domain features with the features of its context [11]), the complexity of meta-programs grows, too. The complexity (in terms of implemented feature variants) may achieve the level at which it is difficult to manage the product (e.g., to introduce changes) due to a diminished understandability.

In this paper, we address the problem of how to manage the complexity and understandability of heterogeneous meta-programs specified using two different languages (meta- and target ones). We suggest the concept of *multi-stage meta-programming* and propose the reverse engineering-based approach to tackle with the task. Our contribution is the higher-level models to analyze and understand meta-programs *per se* and the derivative models enabling to explain and understand multi-stage meta-programs using the proposed approach. The latter can be seen as a theoretical background to develop meta-generators.

The paper is organized as follows. Section 2 analyzes related works. Section 3 introduces the concept of multi-stage meta-programming and describes a reverse engineering approach to extract higher-level models from the meta-program specification. Section 4 presents some properties of multi-stage meta-programs. Section 5 provides a case study to approve the approach experimentally and evaluates complexity of multi-stage meta-programs. Section 6 summarizes and evaluates the approach. Finally, Section 7 formulates conclusions and outlines further research.

2. Related work

The beginning of program comprehension research can be tracked back to the Wirth's proposal in 1968, when he suggested to refuse the use of *go to* statements in a program structure. A few years later, he reformulated the proposal as principles of structural programming. The aim was to improve the structure and understandability in order to make easier the program debugging procedures. Now research in the field is centred on theory, methods and tools [12, 13].

According to the Brooks' theory [12], there are two stages in program comprehension: *hypothesis* and *verification*. Hypothesis is conceptually driven, i.e., the comprehender generates an overall hypothesis about the program's function from its brief description. This high-level hypothesis, combined with programmer's knowledge and expectations, leads to the formation of subsidiary hypothesis about what major structures and operations ought to appear in the program. Verification is both data-driven and conceptually driven. Now working with the program text, the comprehender tries to find support for the hypothesis by looking for elements of a program, called *beacons,* as a mechanism to linking the hypothesis formation stage with the data of the program.

As defined by Brooks [13, 14], beacons are '*key features in a program that serve as typical indicators of a particular structure or operation*'. Beacons serve as interpretive units that signal commonly recognizable elements of a program. Beacons "carry a lot of the meaning" in the code as it is illustrated by an example from a sorting program:

```
temp := a[j];
a[j] := a[i];
a[i] := temp;
```

Other concepts, related to beacons, are *fact finding* [15] and *concept assignment* [16]. As it is very difficult task to recover human concepts embedded into a program text using some tools (e.g., due to *location problem* [16]), there are two general approaches in program comprehension: *human-centred* approach and *tool-based* approach [17]. The first approach focuses more on human behaviour, activities and processes, program cognition models, program readability, commencing, documentation, etc.; whereas the second approach concentrates more on syntactical aspects (e.g., structure, representation, location and identification of features and concerns, slicing/chunking, etc.).

More generally, the human-centred approach can be seen as a software psychology aiming to discover and describe human limitations in interacting with computers. Shneiderman [18], the pioneer of software psychology, defines the topic as the '*study of human performance in using computer and information systems*.' It uses the techniques of experimental psychology to analyze aspects of human performance in computer tasks. It also applies the concepts of cognitive psychology to the cognitive and perceptual processes involved in computer interaction. In software maintenance, e.g., the understanding of human skills and capacity to work with software is necessary in order to facilitate the maintainer's examination and understanding of source code. Software psychologists focus on such human factors as ease of use, simplicity in learning, improved reliability, enhanced user satisfaction, and cultural acceptability [19]. Strengths and limitations of human abilities serve as underlying factors, e.g., in determining the functionality of software maintenance tools.

Many diverse tools exist to assist in program comprehension. The tools can be roughly categorized according to three categories [20]: *extraction*, *analysis* and *presentation.* Extraction tools include parsers and data gathering tools. Analysis tools perform static and dynamic analyses to support activities such as clustering, concept assignment, feature identification, transformations, domain analysis, slicing, and metrics calculations. Presentation tools include code editors, and browsers.

There is some debate between two classes of models: text understanding models and problem solving models [21]. To assess what type of model to use for program understanding, one first should take into account the effect of purpose for understanding (e.g., modification, reuse, debugging or documenting). Another important issue is the type of a program and characteristics of a program under consideration (e.g., size, technology, etc.). As the role of domain-specific languages is constantly growing, the comprehension of such kind of programs is also at the focus [22].

Though the program complexity and understanding are closely related topics, here we restrict ourselves to analysis of the first and recommend a reader to look at the extensive analysis on the program and meta-program complexity issues presented in [23]. We do not analyze the aspects of heterogeneous meta-programming here (an intensive study on that can be found in [9]). Instead, we focus on the concept of multi-stage programming.

Taha, perhaps, was the first who have introduced the concept and presented an extensive study of multi-stage programming in his dissertation [24]. The concept is related to the fundamental principle of information hiding through the introduction of a set of abstraction levels (stages) in order to gain a great deal of flexibility in managing the program construction process. Carette and Kiselyov [25] apply the techniques in the context of homogeneous meta-programming for eliminating the abstraction overhead from generic code. Westbrook *et al.* [26] demonstrate multi-stage programming in an extension of Java, called Ligthweight Java. Megacz [27] also discusses some properties of multi-stage programs (such as generalized arrows within homogeneous meta-programming). The concept is also exploited in other contexts, too. For example, Czarnecki *et al.* [28] use the staged configuration approach to manage multi-level feature models. Rajlich and Bennett [29] discuss a staged model of software evolution. Having in mind the usefulness of multi-stage programming, in this paper we present a variant of multi-stage programming as it is applied to heterogeneous meta-programming.

3. Meta-program understanding and concept of multi-stage meta-programming

We analyze understandability of a given heterogeneous meta-program assuming that it is syntactically and semantically correct. Our approach is based on the human-centred Brook's program cognition theory and reverse engineering. The first gives the basic concepts and enables us to select the strategy in understanding the meta-program, while the second provides the methodology. The result is higher-level models (expressed through beacons and their relationships) and the description of the generation process (algorithm) both extracted from the meta-program specification. Next, we introduce the concept of a multi-stage meta-program and analyze its structure again using the introduced approach. The results of this analysis are high-level models and a process to understand the multi-stage meta-program behaviour.

One important aspect should be emphasized here. A heterogeneous meta-program (see DEF. 1) depends neither on the meta-language type, nor on the domain language type. Usually 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.). This has been proven, e.g., in [30], where Java was used in the role of the meta-language, and in [31], where C++ was used as a meta-language. However, the application domain and designer's flavour are the most decisive attributes for selecting the languages. Recently, we have stated the use of PHP as a meta-language for developing e-commerce product line.

In this paper, to illustrate the basic concepts and to validate the approach by the case study, we have selected Open PROMOL as a meta-language due to the following reasons: 1) it realizes the concept of external meta-functions very similar to those of pre-processing commands such as well-known C (C++) directives; 2) it has a human-readable meta-interface easing to understand the meta-programming per se; 3) the approach and models we deal with here do not depend upon the languages used.

3.1. Basic terms

DEFINITION 1. Heterogeneous meta-program is a specification whose functionality is expressed at two different abstraction levels: the lower-level specifies the basic domain functionality using a domain language, and the meta-level specifies the anticipated variants of the domain functionality using a meta-language.

DEFINITION 2. Beacons are key features in a program or meta-program that *'serve as typical indicators of a particular structure or operation*'. Beacons of a heterogeneous meta-program are meta-parameters within meta-interface and meta-functions within meta-body.

DEFINITION 3. Beacons of the multi-stage meta-program are a meta-parameter within the multi-level interface, and a meta-function within the meta-body, and meta-function label ("\") to deactivate its interpretation when it is executed.

DEFINITION 4. Stage is a *logically and semantically separable* part of a meta-program or a *processing phase* during execution of the multi-stage meta-program.

DEFINITION 5. One-stage meta-program is a software generator. Two-stage meta-program is a meta-meta-program (aka meta-generator). The *k*-stage meta-program is the *k*-meta-program (or *k*-meta-generator).

DEFINITION 6. A meta-function is *active* if it performs its pre-scribed action as a construct of a meta-language at the current stage of meta-program execution.

DEFINITION 7. A meta-function is *passive* if it has been deactivated and is treated as a part of a target language text at the current stage of meta-program execution. For example, '**@sub [..]**' is an active meta-function, '\**@sub[..]**' is a passive meta-function, and '\' is the de-activation label.

DEFINITION 8. Multi-stage meta-program can be derived from the one-stage meta-program by transforming its meta-interface into a multi-level interface, where meta-parameters are distributed among different levels (stages), and by reconstructing its meta-body using the meta-function deactivation mechanism (labels) so that the relationship among stages and meta-functions satisfies some semantic rules and requirements.

DEFINITION 9. Cyclomatic complexity (CC) of a heterogeneous meta-program is the number of different program instances that can be generated from it [23].

DEFINITION 10. Cyclomatic complexity of the *k*-th stage meta-program is the number of different (*k-1)*-stage meta-programs that can be generated from the first.

3.2. Understanding of one-stage meta-programs

The understanding of a meta-program (see an example in Figure 1) is based on two higher-level models: the meta-parameter relationship model (see Figure 1, *a*) and the parameter─meta-function relationship model (see Figure 1, *c*). Both are extracted from the meta-specification by reverse engineering, and represented as bipartite (bi-chromatic) weighted graphs [32].

Though the models describe key relationships among beacons and provide knowledge on the structure of a semantically and syntactically correct meta-program, they say little about the process. The deeper knowledge is gained through the domain program generation process, which is described by the algorithm (see Figure 2).

3.3. Understanding of multi-stage meta-programs

The understanding of the multi-stage meta-program is based on previous models and processes. For this purpose, we present an example in Figure 3 illustrating the two-stage meta-program of the same functionality as the one given in Figure 1. However, the models and the generation processes are different from the previous ones.

```
@- meta-interface of the meta-program
$
"Select a function:"          {AND, OR}  p1:=AND;
"Enter the number of inputs:" {2..4}     p2:=3;
$
@- meta-body of the meta-program
ENTITY GATE_@sub[p1] IS
  PORT (@gen[p2, {,}, {X}, 1]: IN BIT;
      Y : OUT BIT );
END GATE_@sub[p1];
ARCHITECTURE BEH OF GATE_@sub[p1] IS
  BEGIN
    Y <= @gen[p2, { @sub[p1] }, {X}, 1];
END BEH;
```

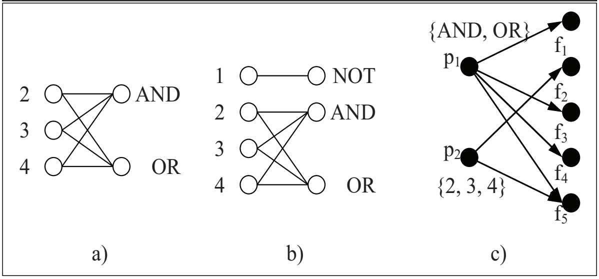

Figure 1. Models obtained through reverse transformation of meta-program: (a) parameter-value dependency; (b) modified parameter relationship; (c) parameter-function relationship

Algorithm of (*k*-1)-stage meta-program generation from *k*-stage meta-program.

Let a semantically and syntactically correct *k*-stage meta-program be given. Then the generation process of the (*k*-1)-stage meta-program is described by the algorithm shown in Figure 4. Note that if we want to generate all possible (*k*-1)-stage meta-programs, we need to repeat actions of the algorithm (steps 3-19) for all possible meta-parameter values at stage *k*.

```
algorithm OneStageMetaprogramming
1.  Identify higher-level beacons (i.e., meta-parameters) within meta-specification;
2.  Specify one value for each meta-parameter;
3.  i:= 1;  /* i - the sequential number of a meta-function within the meta-body */
4.  while i < =m do loop:  /* m - the total number of functions within the meta-body  */
5.        Omit target program fragments (if any) laying before f_i within the meta-body;
6.        Select function f_i  that follows after the omitted fragment in step 5;
7.        Identify parameter-function (f_i) relationships among beacons;
8.        Calculate the value of f_i;
          Substitute the notion of the function f_i by its value within the meta-body;
9.  i := i +1;
10. end loop;
11. Omit the target program fragment (if any) following after f_m;
end algorithm.
```

Figure 2. Algorithm to produce a domain program instance from one-stage meta-program

```
@- first stage interface
$
"Select a function:" {AND, OR}   p1:=AND;
$
@- second stage interface
$
"Set the number of inputs:" {2..4} p2:=3;
$
  ENTITY GATE_@sub[p1] IS
   PORT (\@gen[p2, {, }, {X}, 1]: IN BIT;
         Y : OUT BIT );
  END GATE_@sub[p1];
  ARCHITECTURE BEH OF GATE_@sub[p1] IS
  BEGIN
    Y <= \@gen[p2, { @sub[p1] }, {X}, 1];
  END BEH;
```

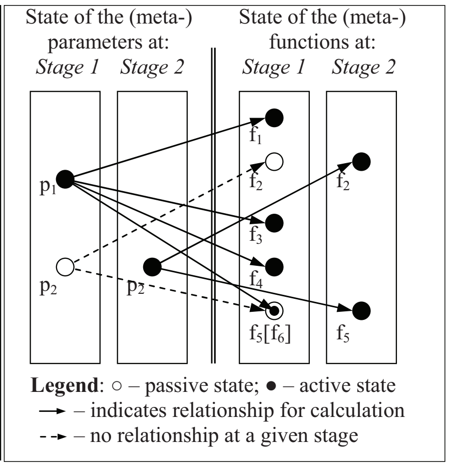
State of the (meta-) parameters at:
Stage 1
Stage 2
State of the (meta-) functions at:
Stage 1
Stage 2
p1
p2
p2
f1
f2
f3
f4
f5[f6]
f2
f5
Legend: ○ – passive state; ● – active state
– indicates relationship for calculation
– no relationship at a given stage

Figure 3. Meta-parameter – meta-function relationship model in two-stage generation process (f_i, i- the short name and sequential number of a function within the text, respectively)

```
algorithm MultiStageMetaprogramming
1.  while k >= 1 then    /* k – number of stages */
2.     read k-stage meta-program;
3.     Interpret the first stage within the k-stage interface by specifying meta-parameter values;
4.     i:= 1; /* i - the sequential number of a meta-function within the meta-body */
5.     while i <= m do loop:
6.        Omit target program fragments (if any) following after f_i within the meta-body;
7.        Select function f_i that follows after the omitted fragment in step 6.
8.        if f_i is active then
9.            Identify relationship among beacons of argument-function (f_i);
10.           Calculate the value of f_i;
11.           Substitute the notion of the function f_i by its value within the meta-body;
12.       else
13.           Exclude interpretation/calculation of f_i;
14.           Remove label "\" standing before f_i ;
15.       end if;
16.       i := i+ 1;
17.    end loop i;
18.    write (k-1)-stage meta-program;
19.    k := k – 1;
20. end loop k;
end algorithm.
```

Figure 4. Algorithm to generate a (k-1)-stage meta-program from k-stage meta-specification

4. Static and dynamic analysis and properties of introduced models

The reverse engineering process is performed as follows. First, we consider the reverse transformation of a one-staged meta-program. The process can be accomplished using static analysis based on the human-centred recognition of beacons within the meta-program, their hierarchy, perception of the relationships among beacons and overall representation. The aim of the analysis is the extraction of a higher-level model. As the graphical representation is the best model for understanding and feature diagrams are widely accepted now, we have selected this notation here. We assume that: 1) a meta-program is well-documented, and 2) the analyst is familiar with the feature diagram notation (see Figure 6).

The result of static analysis might be as it is presented in Figure 5. This model gives the structural (hierarchical) relationship among beacons which are represented as feature hierarchies. Features, in terms of using the feature-based domain modelling approaches (e.g., FODA [33] and its extensions [34]), are externally visible characteristics of a domain. When features are implemented, they represent meta-parameters within the meta-program. What knowledge can be extracted from the feature diagram? The essential knowledge within the diagram is the variability representation described in terms of variant points and variants (aka alternative features). The relationship models among variant points / variants and among variants / meta-functions are presented in Figure 1, *a* and *c*, respectively.

```
(a) @- second stage interface
    $
    "Set the number of inputs:" {2..4}  p2:=3;
    $
    ENTITY GATE_AND IS
    PORT (@gen[p2, {, }, {X}, 1]: IN BIT;
          Y : OUT BIT );
    END GATE_AND
    ARCHITECTURE BEH OF GATE_AND  IS
    BEGIN
      Y <= @gen[p2, { AND }, {X}, 1];
    END BEH;

(b)  ENTITY GATE_AND IS
     PORT (X1, X2, X3: IN BIT; Y: OUT BIT);
     END GATE_AND;
     ARCHITECTURE BEH OF GATE_AND IS
     BEGIN
       Y<= X1 AND X2 AND X3;
     END;
```

Figure 5. Example of VHDL program generated from meta-specification through forward transformation (a) after stage 1, and after stage 2 (b)

The static analysis has a serious restriction: not all types of relationships among features (beacons) can be easily detected and represented since knowledge is woven within meta-functions. Though, e.g., this model (Figure 5) clearly separates beacons of the meta-language (ML) from beacons of the domain language (DL), the understanding of the relationships is difficult and we need to move either to analysis of the process algorithm (see Figure 2) and meta-functions semantics or to use dynamic analysis.

The dynamic analysis is a kind of forward transformation to extract a lower-level model based on the specification execution and analysis of the result obtained. For example, the analysis of the specification (Figure 1) gives the result (Figure 6). Both kinds of transformations (reverse and forward) are valuable for understanding.

Now we are able to consider the multi-stage program understanding. For that, we formulate some properties of the one-stage and multi-stage meta-programs. Meta-function is said to be a simple function if it does not have other functions as argument; otherwise, the function is a compound meta-function (see function f_5 in Figure 3).

PROPERTY 1. All meta-functions within a one-stage meta-program are active; when executed, they return the constructs of a domain language.

PROPERTY 2. A meta-function within a multi-stage meta-program can be either active or passive, as prescribed by the designer.

PROPERTY 3. The passive meta-function may have one or more labels ('\') indicating the number of stages at which the function remains in the passive state. When interpreted, the function is not executed, but its label (one of labels) is removed. After removing the last label, the function becomes active in the next stage.

PROPERTY 4. Relationship model among meta-parameters and their values is the bipartite graph $G((V,R),U)$, where V is a set of meta-parameters values, R is a set of their properties; a set of edges U is a parameter value and its property relationship ($u_{ij} = (v_i, r_j); v_i \in V, r_j \in R; u_{ij} \in U$; $i = 1..|V|$; $j = 1..|R|$). If all parameter values are orthogonal, the bipartite graph $G((V,R),U)$ is complete (see Figure 1, *a*); otherwise, it is a disjoint incomplete bipartite graph (see Figure 1, *b*).

PROPERTY 5. Meta-parameter / meta-function relationship model of the one-stage meta-program is a directed bipartite weighted graph $G((P^w, F), U^*)$, where *P* is a set of meta-parameters, *w* is a vector of the meta-parameter values; *F* is a set of meta-functions; U^* is a set of directed arcs $(p_i, f_j) \in U^*(p_i \in P; f_j \in F)$ (see Figure 1, *c*).

PROPERTY 6. Meta-parameter / meta-function relationship model of the multi-stage meta-program is a decomposition of a directed bipartite weighted graph $G((P_a{}^w, P_p{}^w), (F_a, F_p), U^*)$, where $(P_a{}^w, F_a)$ and $(P_p{}^w, F_p)$ are active and passive parameters and meta-functions, respectively; and the following conditions are valid:

a) $P = P_a \cup P_p; P_a \cap P_p = \varnothing;$

b) $F = F_a \cup F_p; F_a \cap F_p = \varnothing;$

c) PROPERTIES 2 and 3 are valid (see also Figure 3).

PROPERTY 7. If meta-parameters of a multi-stage meta-program within different stages are orthogonal (independent), then Cyclomatic complexity CC_k of the *k*-stage meta-program ($k > 1$) is the product of Cyclomatic complexities of lower-stage meta-programs:

$$CC_k = \prod_{j=1}^{k-1} CC_j .$$

The feature-based model extracted from Figure 1 (see text) as a result of reverse transformation is given in Figure 5. The model describes variability-commonality in terms of parent-child feature relationships and constraints. Though the model specifies higher-level features (beacons) well, the constraints (denoted as R_i) are presented vaguely. The model in Figure 1, *a* and *c* is a more precise representation of this relationship type. It is also difficult to extract the DL and ML relationship using the feature model. Such a relationship can be seen after forward transformation, when DL constructs are fully separated from ML (see Figure 6).

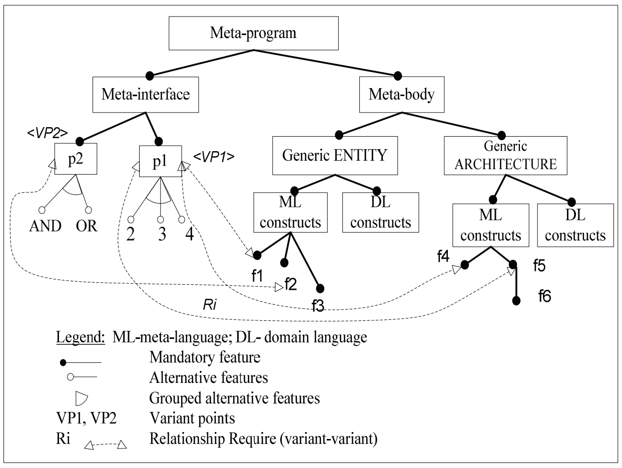

Figure 6. Feature model extracted through reverse transformation for an example given in Figure 1

What is the advantage of using high-level models such as the one given in Figure 5? Its value is the explicit communality-variability modelling and intuitive understanding.

5. Case study and experiments

We analyze two real components of different complexity that were developed as two-stage meta-programs: majority vote function and matrix multiplication.

5.1. Majority vote function

Majority voting is defined as follows. Given a set of values $X = \{x_1, x_2, ..., x_n\}$ and a counting operator $|\ .\ |$, the majority x^{MV} of X is as follows:

$$x^{MV} = x_i, if |x_i| > \frac{n}{2}, i \in \{1...n\}, n - \text{voting order.}$$

The majority vote function is used in different applications such as fault tolerant computing and machine learning. One-stage meta-program for implementing the majority vote function using Open PROMOL [35] as meta-language and VHDL as domain language has been presented in [36]. Here we present a multi-stage version of it (Figure 7), one of its meta-program instances (Figure 8) and a fragment of generated VHDL program instance (Figure 9). The first stage of a multi-stage meta-program has one meta-parameter: the order of voting, and the second stage has another – the width of the data vector.

```
$
"Enter the order of voting"  {3,5,7}         order:=3;
$
$
"Enter data width"          {8,16,24,32}      width:=8;
$

entity MAJORITY is
  port(@gen[order,{, },{X},0]:
         in bit_vector(0 to \@sub[width-1]);
    MAJ_OUT: out bit_vector(0 to \@sub[width-1]);
    IS_MAJ: out bit);
end MAJORITY;

architecture BEH of MAJORITY is
  type arrtype is array (0 to @sub[order/2]) of integer;
  begin
  process (@gen[order,{, },{X},0])
    variable count: arrtype;
    begin
@for[i,0,order/2,{
      count(@sub[i]) := 0;}]
@for[i,0,order/2,{
@for[j,i+1,order-1,{
      if (X@sub[i]=X@sub[j]) then
        count(@sub[i]) := count(@sub[i])+1;
      end if;}]}]
@for[i,0,order/2,{
      if (count(@sub[i])>=@sub[order/2]) then
        MAJ_OUT <= X@sub[i];
        IS_MAJ <= '1';
      else}]
        IS_MAJ <= '0';
@for[i,0,order/2,{
      end if;}]
  end process;
end BEH;
```

Figure 7. Two-stage meta-program of majority vote function

Figure 8 presents one of meta-program instances generated after first stage of forward transformation.

```
$
"Enter data width"    {8,16,24,32}  width:=8;
$

entity MAJORITY is
  port(X0, X1, X2: in bit_vector(0 to @sub[width-1]);
    MAJ_OUT: out bit_vector(0 to @sub[width-1]);
    IS_MAJ: out bit);
end MAJORITY;

architecture BEH of MAJORITY is
  type arrtype is array (0 to 1) of integer;
  begin
  process (X0, X1, X2)
    variable count: arrtype;
    begin
      count(0):=0;  count(1):=0;
      if (X0=X1) then count(0) := count(0)+1; end if;
      if (X0=X2) then count(0) := count(0)+1; end if;
      if (X1=X2) then count(1) := count(1)+1; end if;
      if (count(0)>=1) then
        MAJ_OUT <= X0;  IS_MAJ <= '1';
      else
      if (count(1)>=1) then
        MAJ_OUT <= X1;  IS_MAJ <= '1';
      else IS_MAJ <= '0';
      end if;
      end if;
  end process;
end BEH;
```

Figure 8. Instance of 3rd order one-stage meta-program of majority vote function generated from two-stage meta-program

Figure 9 presents one of VHDL programs generated at the final stage of processing (note that architecture of the VHLD component is the same as in Figure 8).

```
entity MAJORITY is
  port(X0, X1, X2: in bit_vector(0 to 7);
    MAJ_OUT: out bit_vector(0 to 7);
    IS_MAJ: out bit);
end MAJORITY;

architecture BEH of MAJORITY is
...
end BEH;
```

Figure 9. Instance of 3rd order 8-bit width majority vote function (a fragment)

5.2. Matrix multiplication

Matrix multiplication is a binary operation that takes a pair of matrices, and produces another matrix. Matrix multiplication is widely used in image processing and physical modelling applications.

The product of an $n \times n$ matrix A with an $n \times n$ matrix B is an $n \times n$ matrix C:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j} \ ,$$

where $1 \le i \le n$ is the row index, and $1 \le j \le n$ is the column index.

We can extend the matrix multiplication operation to $m, \ m > 1$, matrices assuming that all matrices are of equal size $n \times n$ using recursion:

$$C_{i,j}^{m} = \sum_{k=1}^{n} C_{i,k}^{m-1} \cdot A_{k,j}^{m} \ ,$$

where C^m is the product of matrices $A^1, \ldots, A^m$.

The two-stage meta-program implementing matrix multiplication is presented in Figure 10 (meta-language – Open PROMOL, domain language – C). The number of matrices is defined at the 1st stage of multi-stage interface, and the size of matrices is defined at the 2nd stage.

```
$
"Enter number of the matrices:"     {2..5}   m:=3;
$
$
"Enter size of the matrices:"       {2..8}   n:=2;
$
\@for[i,1,n,{
\@for[j,1,n,{
    C[\@sub[i],\@sub[j]]=@for[z,m-1,1,{@if[z/=m-1,{(}]
\@for[k@sub[z],1,n,{ }]
@for[z,0,m-1,{\
    @if[z>1,{\}])}]@if[z>0,{ * }]\
A@sub[z+1][\@sub[@if[z=0,{i},{k@sub[z]}]],\@sub[
@if[z/=m-1,{k@sub[z+1]},{j}]]]\
@if[z/=0,{\@if[k@sub[z]/=n,{ + \}]}]
}]}];}]}]
```

Figure 10. Two-stage meta-program of matrix multiplication

After the 1st processing stage, meta-language processor generates meta-programs for generating multiplication of m matrices (see an instance for 3 matrices in Figure 11). Note that in this case, not only domain language code, bus also parts of meta-language code are generated.

```
$
"Enter size of the matrices:"       {2..8}   n:=2;
$
@for[i,1,n,{
@for[j,1,n,{
    C[@sub[i],@sub[j]]=
@for[k2,1,n,{ (
@for[k1,1,n,{
    A1[@sub[i],@sub[k1]]
     * A2[@sub[k1],@sub[k2]] @if[k1/=n,{ + }]
    }]) * A3[@sub[k2],@sub[j]] @if[k2/=n,{ + }]
}];}]}]
```

Figure 11. Instance of one-stage meta-program for multiplication of 3 matrices generated from two-stage meta-program

Finally, Figure 12 presents an example of code in C generated for multiplication of 3 matrices of 2×2 size.

```
C[1, 1]= ( A1[1, 1] * A2[1, 1] +
           A1[1, 2] * A2[2, 1] ) * A3[1, 1] +
         ( A1[1, 1] * A2[1, 2] +
           A1[1, 2] * A2[2, 2] ) * A3[2, 1];
C[1, 2]= ( A1[1, 1] * A2[1, 1] +
           A1[1, 2] * A2[2, 1] ) * A3[1, 2] +
         ( A1[1, 1] * A2[1, 2] +
           A1[1, 2] * A2[2, 2] ) * A3[2, 2];
C[2, 1]= ( A1[2, 1] * A2[1, 1] +
           A1[2, 2] * A2[2, 1] ) * A3[1, 1] +
         ( A1[2, 1] * A2[1, 2] +
           A1[2, 2] * A2[2, 2] ) * A3[2, 1];
C[2, 2]= ( A1[2, 1] * A2[1, 1] +
           A1[2, 2] * A2[2, 1] ) * A3[1, 2] +
         ( A1[2, 1] * A2[1, 2] +
           A1[2, 2] * A2[2, 2] ) * A3[2, 2];
```

Figure 12. Instance of matrix multiplication code

The meta-meta-component, in comparison to its counterpart meta-component, is a more flexible specification (in terms of the capability to manage complexity) with distributed functionality that is generated on demand dependent upon the context of use.

Complexity of the meta-program is distributed within stages when the multi-stage approach is applied. We have applied the number of meta-functions and Cyclomatic complexity as a metric of meta-program complexity [23]. The results are presented in Table 2.

Table 1. Complexity of developed multi-stage meta-programs

Case study	No. of meta-functions at 1st stage	CC of 1st stage	No. of meta-functions at 2nd stage	CC of 2nd stage	Total CC
Section 5.1	16	3	2	4	12
Section 5.2	13	4	14	7	28

6. Discussion and evaluation of the approach

We have analyzed the understandability of heterogeneous meta-programs. To deal with the problem, we have applied the reverse engineering approach combined with ideas of the Brook's program cognition theory resulting in the extraction of higher-level models from the correct executable meta-specification (i.e., meta-program). The understanding of these models has enabled us to suggest the concept of multi-stage meta-programs and to analyze them using the same approach.

Structurally, multi-stage meta-programs are derived from one-stage meta-programs. Conceptually, multi-stage meta-programs are constructed through the decomposition of the meta-level into separate sub-levels of abstraction, called *stages*. To preserve semantic constraints, the decomposition has to be supported by a meta-function de-activation mechanism that changes the role of the meta-function from active to passive and vice versa at a particular meta-program execution stage.

Multi-stage programs are applicable in the following cases: 1) when the variation of domain features is large enough and we need to manage complexity issues (in order to avoid the component over-generalization problem [37] when using meta-programming); 2) when we need to achieve a higher degree of flexibility in managing groups of components that contain derivative features, e.g., in Product Line Engineering; 3) when we need to manage the library scaling problem [38].

From the pure methodological viewpoint, all these cases can be seen as a contribution to the design of meta-generators (i.e., generator generators).

The disadvantages of the proposed approach: maturity is not enough; reverse engineering is subjective: tools are needed, but it is little known about the use of the tools. It is not clear at what extent other meta-languages can support the concept of multi-stage meta-programming. The approach specifies well the only one side of the problem – the extraction of higher-level models from the correct specification thus contributing to its understandability. However, meta-program understanding is also related with program changeability and evolution. In this case, a synergetic approach should be applied, in which the reverse and forward transformation approaches are to be combined together in order to be able not only to understand the meta-program itself and required changes, but also to implement changes and approve their correctness.

7. Conclusions and further work

Though there is a wide front of research in the field of program comprehension, it is still little known about the understandability and complexity issues of meta-programs. We have shown that the models constructed to better understanding of heterogeneous meta-programs *per se* can disclose new useful features such as a multi-stage structure within the given meta-program and its multi-stage processing.

In general, the value of multi-stage meta-programming is seen as a contribution to the theoretical background of meta-program transformations to develop meta-generators using meta-programming. Another value of the proposed approach is the opportunity to manage complexity issues easily (i.e., through the distribution of meta-parameters among stages) in developing and using heterogeneous meta-programs.

As the maturity level of meta-program comprehension is still low, further work on the equivalent transformations of multi-stage and multi-linguistic meta-programs as well as their applications is needed. We intend to use the approach for the development of the platform to design collective e-commerce systems (web application domain) and generative learning object (GLO) design (e-learning domain).

Acknowledgements

This research has been partially supported by the Lithuanian Agency of Science, Innovations and Technology via the e-CROWD project.

References

[1] **S. L. Peeger.** *Software Engineering: Theory and Practice*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[2] **P. Hallam.** What Do Programmers Really Do anyway? Microsoft Developer Network (MSDN) C# Compiler, Jan 2006.

[3] **K. Kang, J. Lee, P. Donohoe.** Feature-oriented Product Line Engineering. *IEEE Software*, 19 (4), 2002, 58–65.

[4] **W. B. Frakes, K. Kang.** Software Reuse Research: Status and Future. *IEEE Transactions on Software Eng.,* 31(7), 2005, 529–536.

[5] **M. Garg, M. K. Jindal.** Reverse Engineering – Roadmap to Effective Software Design. *Int. Journal of Recent Trends in Engineering*, 1(2), 2009.

[6] **M. Petrenko, V. Rajlich, R. Vanciu.** Partial Domain Comprehension in Software Evolution and Maintenance. In R.L. Krikhaar, R. Lämmel, C. Verhoef (Eds.), *16th IEEE Int. Conf. on Program Comprehension, ICPC 2008,* Amsterdam, The Netherlands, June 10-13, 2008, 13–22.

[7] **J. P. Gibson, J. O'Kelly.** Software Engineering as a Model of Understanding for Learning and Problem Solving. *Int. Computing Education Research Workshop* (*ICER'05),* October 1–2, Seattle, Washington, USA, 2005, 87–97.

[8] **T.L. Veldhuizen.** Tradeoffs in Metaprogramming. Proc. of ACM SIGPLAN *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. Charleston, SC, USA, 2006, 150–159.

[9] **R. Damaševičius, V. Štuikys.** Taxonomy of the Fundamental Concepts of Metaprogramming.

Information Technology and Control, 37(2), 2008, 124–132.

[10] **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, 105–114.

[11] **H. Hartmann, T. Trew.** Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains. *Proc. of 12th Int. Software Product Line Conf., SPLC '08*, 8-12 Sept. 2008, 12–21.

[12] **R. Brooks.** Towards a Theory of Computer Program Comprehension. *Int. Journal of Man-Machine Studies,* Vol. 18, 1983, 543–554.

[13] **M.A. Storey.** Theories, Methods and Tools in Program Comprehension: Past, Present and Future. In Proc. of the 13th Int. Workshop on Program Comprehension (IWPC'05), IEEE, 2005, 181–191.

[14] **S. Wiedenbeck.** Beacons in Computer Program Comprehension. Int. Journal of Man-Machine Studies, 25, 1986, 697–709.

[15] **T. D. LaToza, D. Garlan, J. D. Herbsleb, B .A. Myers.** Program Comprehension as Fact Finding. Proc. of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, ESEC-FSE'07, September 3–7, Croatia, ACM, 2007, 361–370.

[16] **T. J. Biggerstaff, B. W. Mitbander, D. Webster**. The concept assignment problem in program understanding. Proc. of the 15th International Conference on Software Engineering, 1993, 482–498.

[17] **S. Rugaber.** Program Comprehension. Encyclopedia of Computer Science and Technology, 1995.

[18] **B. Shneiderman.** Software Psychology: Human Factors in Computer and Information Systems. Little Brown, 1980.

[19] **R. Damaševičius.** On The Human, Organizational and Technical Aspects of Software Development and Analysis. In Papadopoulos, G.A., Wojtkowski, W., Wojtkowski, G., Wrycza, S., Zupancic, J. (Eds.), Information System Development: Towards a Service Provision Society. Springer, 2009, 11–19.

[20] **S. R. Tilley, D. B. Smith.** Coming Attractions in Program Understanding. Technical Report CMU/SEI-96-TR-019, 1996.

[21] **F. Détienne.** What Model(s) for Program Understanding? UCIS'96, Colloque Using Complex Information, Poitiers, France, September 4-6, 1996.

[22] **M. J. V. Pereira, M. Mernik, D. da Cruz, P. R. Henriques.** Program Comprehension for Domain-Specific Languages. Journal on Computer Science and Information Systems, 5(2), 2008, 1–17.

[23] **R. Damaševičius, V. Štuikys.** Metrics for Evaluation of Metaprogram Complexity. Computer Science and Information Systems (ComSIS), 7(4), 2010, 769–787.

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

[25] **J. Carette, O. Kiselyov.** Multi-stage Programming with Functors and Monads: Eliminating Abstraction Overhead from Generic Code. In R. Glück, M.R. Lowry (Eds.): Generative Programming and Component Engineering, 4th Int. Conf., GPCE 2005, Tallinn, Estonia, Springer 2005, 256–274.

[26] **E. Westbrook, M. Ricken, J. Inoue, Y. Yao, T. Abdelatif, W. Taha.** Mint: Java multi-stage programming using weak separability. Proc. of ACM SIGPLAN Conf. on Programming language design and implementation (PLDI '10). ACM, NY, USA, 2010, 400–411.

[27] **A. Megacz.** Multi-Stage Programs Are Generalized Arrows. The Computing Research Repository (CoRR) abs/1003.5954, 2010.

[28] **K. Czarnecki, S. Helsen, U. Eisenecker.** Staged Configuration through Specialization and Multi-Level Configuration of Feature Models. Software Process Improvement and Practice, 10, 2005, 143–169.

[29] **V. T. Rajlich, K. H. Bennett.** The Staged Model of the Software Lifecycle. IEEE Computer, July 2000, 66–71.

[30] **V. Štuikys, R. Damaševičius**. Metaprogramming 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, 229–250.

[31] **R. Damaševičius, V. Štuikys**. High Level Design of Soft IPs using C++ and SystemC. Information Technology and Control, 4(25), 2002, 54–64.

[32] **R. Diestel**. *Graph Theory* (3rd ed.), Springer, 2005.

[33] **K. Kang, S. Cohen, J. Hess, W. Novak, S. Peterson.** Feature-Oriented Domain Analysis (FODA) Feasibility Study. TR CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

[34] **P.-Y. Schobbens, P. Heymans, J.-Ch. Trigaux**. Feature Diagrams: A Survey and a Formal Semantics. *Proc. of the 14th IEEE Int. Requirements Engineering Conf.,* September 11 - 15, Washington, DC, 2006, 136–145.

[35] **V. Štuikys, R. Damaševičius, G. Ziberkas.** Open PROMOL: An Experimental Language for Target Program Modification. In A. Mignotte, E. Villar, L. Horobin (eds.), *System on Chip Design Languages*. Kluwer Academic Publishers, Boston, April 2002, 235–246.

[36] **V. Štuikys, R. Damaševičius, G. Ziberkas, G. Majauskas.** Soft IP Design Framework Using Metaprogramming Techniques. In B. Kleinjohann, K. H. (Kane) Kim, L. Kleinjohann, A. Rettberg (eds.). *Design and Analysis of Distributed Embedded Systems.* Kluwer Academic Publishers, Boston, 2002, 257–266.

[37] **J. Sametinger.** *Software Engineering with Reusable Components*. Berlin: Springer, 1997.

[38] **T. Biggerstaff.** The Library Scaling Problem and the Limits of Concrete Component Reuse. *Proc. of the Third Int. Conf. on Software Reuse: Advances in Software Reusability*, Rio de Janeiro, Brazil, 1994, 102–109.

Received April 2011.