ISSN 1392 – 124X INFORMATION TECHNOLOGY AND CONTROL, 2009, Vol. 38, No. 2

DEVELOPMENT OF WEB COMPONENT GENERATORS USING ONE-STAGE METAPROGRAMMING

Vytautas Štuikys, Marijus Montvilas, Robertas Damaševičius

Software Engineering Department, Kaunas University of Technology
Studentų St. 50, LT–51368 Kaunas, Lithuania
e-mail: vytautas.stuikys@ktu.lt, marijus.montvilas@internet.ktu.lt, damarobe@soften.ktu.lt

Abstract. We consider a methodology for the development and application of a class of generators that are externally parameterized tools enabling to generate Web component instances on demand depending on the context of use. Such generators are generalized entities of conventional Web components that indeed are lower-level generators for the portal domain. We use one-stage heterogeneous metaprogramming techniques for implementing the externally parameterized metaprograms as a specification of the generators. The first our contribution is a systemized process to create the externally parameterized metaprograms for building Web domain generators. The process describes a logical linking into the coherent structure of the following entities: semantic model for change, program generator model, Web component instance model, and given metalanguages. The second our contribution is the complexity estimation of Web component generators that were developed and used for generating Web component instances to incorporate them into real portal settings. The complexity is estimated using the Kolmogorov's complexity measures and Cyclomatic Complexity. We analyze also specific features and characteristics of the developed generators.

Keywords: Web component models, Web component generator, one-stage heterogeneous metaprogramming, complexity measures of metaprogram, Kolmogorov's complexity.

1. Introduction

Over at least ten past years, various organizations have envisioned portal solutions (Web) as a necessity to develop and maintain integrated, personalized environments for collaborative activities. Though now the rate of growth of Web portals has stabilized in comparison to the boom of the last decade of the 20th century [1], creating new portals remains as actual problem as ever due to unsurpassable capabilities of Web technology for connecting peoples for interaction and information interchange. Many organizations and design teams are involved in creating Web-based applications until now. This continues to happen because of the fact that the initial Web implementation, defined by its static nature and a purposefully low barrier to entry, was sufficient only for some time for sharing documents. But now this is inadequate to more advanced applications.

Though the infrastructure problems of the Web have been largely solved now [2-4], the market pressure and complexity growth of applications result in the need for a more effective design process of the Web portal development per se. The problem we deal with here is the development of Web component generators through the use of the one-stage metaprogramming techniques.

In the context of the paper, by Web component generators we mean a class of generators which are externally parameterized tools enabling to generate Web component instances on demand depending on the context of use. Such generators are generalized entities of conventional Web components that indeed are lower-level generators for the portal domain [5-7]. Before using in a concrete context, parameterized generators are firstly to be instantiated.

By one state-stage metaprogramming techniques we mean heterogeneous metaprogramming [8], in which the functionality of portal domain is expressed through multiple languages relevant to the domain and the external generalization is expressed using at least one metalanguage (e.g., ASP that is used as a metalanguage to express through parameterization scripting and modifications). The term "one-stage" means that we do not exclude the possibility to apply "multi-stage" metaprogramming in the other context. The aim of using the metaprogramming techniques is to enhance reuse and to extend generating capabilities of the known technological infrastructure, such as the one proposed by Microsoft [9] and used in practical implementations of the created generators.

The first our contribution is a systemized process to create the externally parameterized metaprograms for building Web domain generators. The process describes a logical linking of the following entities into the coherent structure: semantic model for change, program generator model, Web component instance model, and given metalanguages. The second our contribution is the complexity estimation of Web component generators that were developed and used for generating Web component instances to incorporate them into real portal settings. The complexity is estimated using the Kolmogorov's complexity measures [10] and Cyclomatic Complexity.

The rest part of the paper is organized as follows. Section 2 analyzes related works. Section 3 presents a domain analysis framework and portal domain models (as a result of the analysis and input to building generators). Section 4 provides the problem formulation and the definition of basics terms. Section 5 analyzes some properties of Web components. Section 6 presents the Web component generator model. Section 7 describes a method as a detailed process for creating externally parameterized metaprograms, i.e., input specifications that enable functioning of domain generators. Section 8 provides and deals with experimental results and complexity issues of the developed generators. Section 9 provides analysis and evaluates the total results. Finally, Section 10 states conclusions and future work.

2. Related Works

Since we consider the development of portal components as a connection of two domains (i.e., portal technology and its application, and solution domain based on using meta-programming techniques), we identify two basics streams of related works: 1) Web technology and Web-based generators; 2) heterogeneous metaprogramming techniques. Other sources which are important in the context of the paper (such as those related to domain analysis and reuse) are introduced in other Sections together with our considerations.

Stream 1. As the portal development is indeed a very wide field, here we restrict ourselves by presenting only those works which are either general and most informative for portal development (such as overviews, taxonomies, technology needs evaluation, etc.) or directly relates to the problem we consider (such as Web-based generators).

Coffmam and Odlyzko [1] evaluate the size and the growth rate of the Internet at the end of the last century when the boom of the technology was evident. Fielding and Taylor [2] analyze design issues of the modern Web architecture. In a similar paper, Hazra [3] analyzes architectures of Enterprise Portals and formulates basic principles of their design. ST Electronics (Info-Software Systems) Pte. Ltd. [11] applies a lightweight, reactive approach to support an industrial Web Portal product line. According to the announcement, unique characteristics of the approach are fast, low-cost migration from a single, conventional Web Portal towards a reusable "generic Web Portal" solution, effective handling of large number of functional variants and their dependencies, the ability to rapidly develop new Web Portals from the generic one, and to independently evolve multiple Web Portals without ever losing a connection between them and the "generic Web Portal".

Doyle and Lopes [4] present a survey of Web based technologies for the Web application development. Authors conclude that although the infrastructure problems have largely been solved, "the cacophony of technologies for Web-based applications reflects the lack of a solid model tailored for this domain". Losh [7] analyzes the simplest Web generators as a form of the online hypertext and provides taxonomy of some of the most popular generator types (i.e., those that create original verbal or visual online texts).

Rajapakse and Jarzabek [12] identify important technological needs in relation to reference architecture for Web Applications and show how different technological trends address each need. The paper is interesting to those who want to get a grasp of the Web technology landscape and understand major trends. Djemaa *et al.* [6] describe a generator for adaptive Web applications called GIWA that aims at the design and automatic generation of adaptable Web interface. The GIWA methodology is based on three levels: semantic level, adaptation level and presentation level. The implementation of GIWA is based on java swing interface to instantiate the models which are translated in XML files. The methodology uses then XSL files to generate the HTML page corresponding to the user. Helman and Fertalj [5] present generators in Web development process and overview of key features of Web applications and development. The research focuses on how those features are implemented and supported by different Web application generators and other Web development tools.

Stream 2. Metaprogramming-based techniques and approaches are also widely discussed in the literature. What is important to know is that the relative approaches very often are called using different terms (e.g., transformation-based, generative, multi-stage programming, etc.) as it is identified in [13]. In this paper, Damaševičius and Štuikys analyze also the known taxonomies in the field and present a more extended taxonomy of the fundamental concepts of metaprogramming. Taha, in his contributory work [14], calls metaprogramming as 'multi-stage programming'. Trujillo with colleagues [15] generalize the metaprpogramming approaches identifying them as 'generative metaprogramming'. Cruz *et al.* [16] analyze the role of the languages and compare generators for language-based tools.

In general, analysis of the approaches can be categorized as follows: (a) metaprogramming-based approaches (as a solution domain) for generator design (e.g., generative programming [17], aspect-oriented programming [18], metaprogramming techniques [8, 19]); and (b) product line approaches [20-22] in which the main focus is given to variability aspects in order to implement domain generators. Combining approaches in both categories (a) and (b) one might agree with the Veldhuizen's vision that metaprogramming is "the study about software generalization" [23]. We exploit the observation given to the metaprogramming techniques as it is explained in the remaining Sections 3 to 7.

3. Domain analysis

With the generative reuse approach [24, 25] in mind, the initial assumption as a roadmap for investigation is as follows: the approved way for building domain generators is to start from domain analysis. The next assumption follows: either a designer/analyzer is an expert in two domains at once (Web portal and program generator design using metaprogramming techniques), or a team that performs the work has the stated knowledge.

In general, analyzer has at least three possibilities: 1) to apply an ad hoc analysis method; 2) to apply a systematic method which should be selected from a series of known methods such as FODA [26], FORM [27], Prieto-Diaz [28], etc.; 3) to modify the known methods or combine them in somewhat way. Here we use ad hoc analysis. Reasons for that are two: simplicity and our determination to rely on the analyzer's knowledge in the portal field. A success of the use of ad hoc methods depend on such factors as: 1) understanding by analyzer the aims of analysis including the awareness on which aspects to focus; 2) expert's /analyzer's knowledge including literature studies; 3) understanding what result from analysis should be obtained and what representation of the result is most relevant for generators design.

Analysis of (a) generator design approaches [8, 17-19] and (b) product line approaches [20-22] and other sources shows that the main focus should be given to variability aspects. On the other hand, as Web based approaches are so rapidly evolving, the "cacophony of technologies" [4] is another aspect to focus on. We focus on these aspects in analysis of the portal domain. The variability aspects can also be treated as *design for change* [29]. Thus the result of analysis should reflect the both sides of the domain: technological characteristics and application or user-related characteristics. The later issues are conceived in this context as *requirements for change*. Domain analysis results in collecting of the relevant information in order to achieve the goal. In reuse literature (see [30]), this information is called by general term, namely *domain model*.

What aspects describe the domain model we extract through analysis of the portal domain? Firstly, we obtain the portal model that describes technological characteristics which are most important to our context: typical Web components that practically appear in most types of portals, technological infrastructure that is needed to implement portals; domain languages as a part of the infrastructure. The language aspect is represented separately in the model because of the importance of this aspect for generators (most types of generators are language-based tools [16]). As portal domain is the one, which is specified through a multi-linguistic approach, the role of each language within the framework should be identified too. *Table 1* summarizes technological aspects of the domain model.

Table 1. Basic technological aspects of portal domain

Typical Web Components	Technologic frameworks	Language support	Role of languages
Data management (DMG); Content management (CMG); Dynamic module management (DMMG)	ASP,COM, JScript, VB, .net	VB,	For ensuring Web security support aspects
		SQL	For expressing Web component links with DB for generation objects (tables, procedures, functions)
		XML	For storing structural data in server side (alternative for DB)
		XSL	For representing data in format XML on client side
		HTML	For representing text documents in browsers
		CSS	For representing content described in HTML format on client PC
		JScript	For modelling and representing
		ASP	For scripting multi-linguistic files and managing manipulations

The next part of the model is the application-related or possible user-related information. This information connects the anticipated requirements for change with the portal components characteristics. The list of component characteristics with examples includes: data formats, data types, modes of their representation, selecting conditions, features such as a structure of files that are increments of the components, etc. Constraints identify what relationships are not feasible among variants of various characteristics (some of these characteristics are presented along with experimental results in Section 8).

4. Problem statement

The initial data as a preliminary assumption to formulate the problem are as follows: 1) requirements (including constraints) to support *design for change*; 2) Web component generator model; 3) an instance of Web component to be generalized; 4) a metaprogram model and a metalanguage to support one-stage programming. The design task is formulated as follows:

To transform the given models 2 and 3 into the external metaprogram written using the given metalanguage (model 4) so that two pre-defined conditions are satisfied: a) the requirements for change and constraints are fulfilled; b) the metaprogram, when executed, generates a set of Web component instances that are syntactically and semantically correct in the given context of use.

To facilitate the problem to be solved, one preliminary step should be done firstly: we need to combine the requirements for change and models 2 and 3 into a unified model, which we call a *semantic model.* If this preliminary step is done, then the task can be reformulated as a transformation of the semantic model into a metaprogram satisfying the same constrains. The other note relates to the terminology. As there are many terms with a very close meaning, we define the terms with the extending explanation of their meaning in this Section.

(*Parameterized*) *Web component generator* is an externally parameterized tool enabling to generate component instances on demand (the definition emphasizes the process aspects). Such a generator is a generalized entity of conventional Web components that indeed are lower-level generators for the portal domain. Before using parameterized generators in a concrete context, they are firstly to be instantiated. The output of the parameterized generator is not the data (Web page as it is in the case of simple generators) but a domain program that generates Web-based data.

One-stage metaprogramming is the one that uses only one level for expressing generalizations in the heterogeneous metaprogramming paradigm. *External metaprogram* is the one which: a) is developed according to the principles of heterogeneous metaprogramming; b) serves as an input specification to the parameterized generator. *Metaprogram* is also a generator but metaprogram expresses specification aspects that govern the generation process. Metaprogram can be also called *metaspecification.* Structurally, metaprogram (metaspecification) consists of the two interrelated parts: *metainterface* (for specifying parameterization) and *metabody* (for expressing generalizations through modifications/changes).

Web component instance is an object for implementing generalizations, i.e., for creating metaprogram/parameterized generator. *Metalanguage* is a language, which serves for specifying operations that implement generalizations through changes. *Target (or domain language)* is the one which describes some functional aspects of Web component instances.

In the next two Sections, we describe and analyze the structure and some properties of Web component instances, which are essential to understand the instance and generator models, respectively.

5. Properties of Web component instances

As a result of analysis, three typical Web components were identified (see *Table 1*). They are: Data management (DMG); Content management (CMG) and Dynamic module management (DMMG). The components are applied across multiple applications but with slightly different characteristics. Differences follow from the model (see *Table 1).* The next description identifies the properties of components in detail.

1. Structural properties of Web components are as follows: structure of the component is a set of different files each representing some particular attributes (e.g., representation, selection of data, data management, etc.).
2. Components are reusable entities which were not created from scratch but were extracted from the previously developed projects. The latter means that the entities were reused and their quality was approved by the use cases.
3. Files as constituents of the components have the following structure: invariant and variant parts. The invariant part represents the description, which does not depend on the context of use. The variant part represents such a functionality that may vary depending on the context of use, but in a concrete context it has a pre-defined value.
4. When the variant part within the given file is recognized and each variant is identified by a designer, such a structure can be treated as a template for reusing in multiple contexts.
5. As a template file is a model for the instantiation of the file in a given context, it can be treated as a low-level semi-automatic generator; where the instantiation is performed manually and the invariant part is represented automatically (this is a lower-level domain generator [4]).
6. As the intrinsic features of the files describe a variety of specific attributes of the domain (e.g., representation, transferring, security, etc.), files are to be combined into a coherent structure that, from the designer's viewpoint, is treated as a monolithic component.
7. Each file has a separate technological support, i.e., a file is described using different languages (one or a few for the same file, e.g., ASP+ Jscript + HTML). The capabilities of the given technology ensure the composition of files into the coherent monolithic structure through a simple integrating mechanism, i.e., the file name (if there is nothing to transfer to other file) or the call-type statement (if there are data/parameters to transfer to other file). The file name/the call-type statement are also treated as an *interface* for integrating files into a coherent structure.
8. Some files may appear in different components (e.g., some files from DMG are needed to use in CMG) too.

9. The sequencing of files within a given component is arbitrary. The given sequence pre-specifies the structure of the component. As the arrangement of files within the component is arbitrary, one can consider the model of the component as distributed files. The position of the file within the component is also its interface.

The specified properties are sufficient to generalize the Web component model in order to device the externally parameterized Web component generator, which uses the *white-box reuse model* combined with metaprogramming techniques as it is described in the next two sections.

6. Web Component generator model

The aim is to show that typical component models used in other domains and Web-based component models are slightly different entities. Typical component models (e.g., those that describe software [30], or hardware [31] components) contain two essential parts: interface and functionality. The basic feature of the parts is that they are explicitly separable (see Figure 1, *a*). In order to construct a generator based on heterogeneous metaprogramming techniques the component model is to be generalized. One way to do so is to introduce new functional aspects into the initial model through *changes*. The additional functionality usually affects the internal structure of both parts of the model, its interface and functionality. Such a kind of generalization is also known as widening [30]. The generalization using widening results in the creation of a metamodel (metacomponent); this contains metainterface for managing changes and generalized functionality (aka metafunctionality or metabody) for managing generation (when implemented) as it is shown in Figure 1*b*. Metainterface is clearly separable from its metabody (see a dotted line in Figure 1*b*).

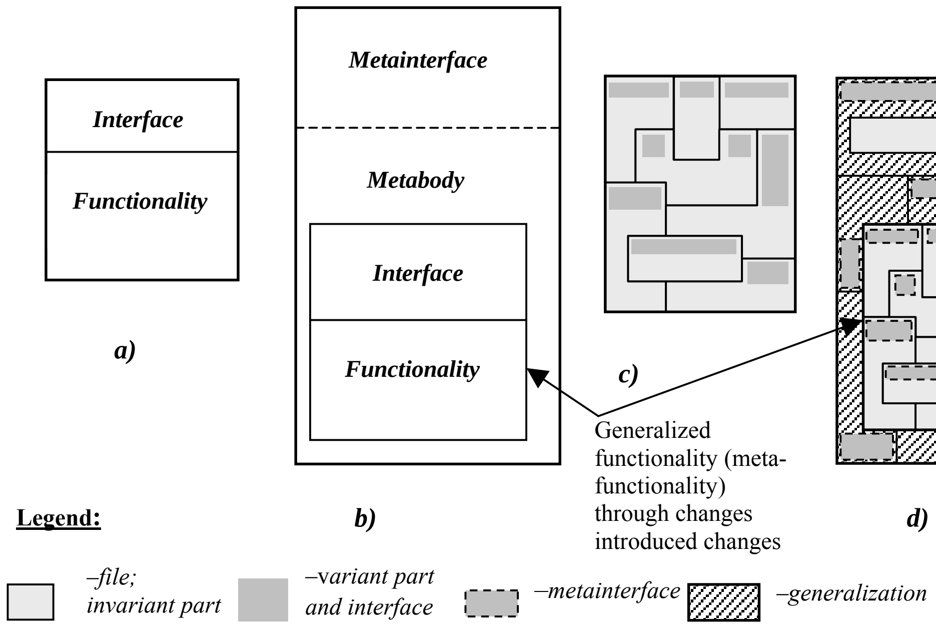

Figure 1. Component models: a) typical component instance; b) typical metacomponent; c) Web-based component model; d) Web-based metacomponent model

Now we can present and discuss the Web-based component and their generator models (see Figure 1*c* and *d*), which are based on the properties discussed in the previous section. The graphical notation has the following meaning. In Figure 1 *c*, contour (rectangular) represents a file; darkened places within a rectangular denote variable part and interface; the rest part of a rectangular denotes the invariant part. In Figure 1 *d*, broken line denotes metainterface as an irregularly distributed structure; the distributed rectangular (with 45 degree lines) indicates generalization aspects (scripting and modification of the files).

Conceptually (if we ignore a great number of files and their distribution within the models *c*, and *d*) models have the same structure as their counterparts (see Figure 1*a* and *b*). The difference can be obtained if one looks at the semantics (roles) and internal structure of these two kinds of the models. The models (*a* and *b*) are intended to describe computational or structural aspects only, while the models (*c, d*) are intended to describe representational, distributional (transferring) and manageable aspects (e.g., to support both server and user sides connections via the network) in the first place. If computational aspects are needed, they are not ignored too, but their role is miserable in this context. It is the reason why we need to reflect all these aspects in the Web-based model (see Figure 1 and compare *a* and *c*). Generalization of the Web-based model (in order to obtain the model *d*) follows the same track as it was described above.

The basic idea behind building domain generators is the model transformation into relevant programs and metaprograms. To implement a domain generator, such as the Web-based generator, a technological support (e.g., Web-based languages and their interpreters) should encompass all aspects the model describes. Since the Web-based component model reflects many aspects of the domain it is difficult to express these aspects using one or two languages only and a series of languages are used (see, e.g., [4, 9] and as well experimental results here). Today, despite of the "cacophony of technologies", the Web infrastructure is developed and it is giving a good enough support for designers.

7. One-stage metaprogramming: A method for implementing domain generators

We accept the following pre-conditions: 1) All initial data are given as they were identified in Section 4; 2) target (domain) language(s) and tools that support the language(s) are execution-level domain generator(s) [6]; 3) metaprogram that modifies in some way a target program written in the target language is the designer's-level (construction time) program generator. The aim is to describe a method stating how metadesigner creates a metaprogram in general, i.e., independently on the application domain and metalanguage. But when we need to make some emphasis on domain specificity, which affects metaprogramming, we reflect the specificity in the method. Below we describe the method as a logical sequence of the processes metadesigner performs in order to develop a metaprogram.

1. To make a choice in selecting a metalanguage or metalanguages (e.g., in Web component design) to support the implementation of generation/generalization aspects (in other words white-box reuse). For example, in the case of using metaprogramming to develop Web-component generators, we apply two metalanguages (e.g., ASP+VB, where ASP supports modification (change) aspects and VB supports Web security aspects) in the same metaspecification at once.
2. To analyze the selected metalanguage/metalanguages and, according to structural programming principles, to form three clusters of constructs for each metalanguage as follows. The first cluster ***Ko*** contains constructs that correspond to operations such as assignment, Read/Write (see Table 2). The second cluster ***Ka*** contains those constructs that enable to realize alternative decisions (e.g., if, case). The third cluster ***Kc*** contains constructs that enable to code loops in a metaprogram. Table 2 provides clusters ***Ko***, ***Ka*** and ***Kc*** for three different languages (Open Promol, JScript, ASP), which may be candidates for the metalanguage selection in various contexts.
3. To perform semantic analysis of requirements for change aiming: a) to identify and fix constraints (if yet they are not given) within requirements; b) to form two clusters (classes) within requirements (*orthogonal changes*, which have no other changes within and *hierarchic changes* that contain other changes within).

Table 2. Clusters as subsets of constructs of three different languages

Change category*	***Operation, cluster name***	Open Promol	JScript	ASP
1st	*Operation,* ***Ko***	*Interface*$ **...$** @**sub**[…]	**Document.Write** (...) **...= ...**	**Response.Write (**...) **…= …**
2nd	*Alternative****, Ka***	@**if**[...] @**case**[...]	**If ...** **Switch/Case ...**	**If ...** **Switch/Case...**
3rd	*Cycle,* ***Kc***	@**for**[...] @**repeat**[...] @**gen** [...]	**For**... **While**...	**For**... **While**...

Note: *) type of change in orthogonal category

What is important to emphasize is that orthogonal changes are also categorized into three categories as follows:

I. Changes of the 1st category are those that are implemented using operations from cluster ***Ko.***
II. Changes of the 2nd category are those that are implemented using operations from cluster ***Ka***.
III. Changes of the 3rd category are those that are implemented using operations from cluster ***Kc.***

Simple examples explain the meaning of orthogonal changes in each category as well hierarchic changes below (see Table 3).

4. To form a semantic model ***SM*** for change, which describes links or relationships among requirements and other given initial data, clusters of operations, constraints and represents changes types by parameters and their values. The model is described by sixth sets as identified by (3):

SM = (*<semantic description for change>, <class of change with operation type>, <labelled component instance>, <change parameter>, <set of parameter values>, <constraints>*) (3).

Table 3. Matching between categories of change and operation clusters and implementation of requirements for change

Category of change	**Requirements for change**	**Object for change**	**Cluster/ operation**	**Implementation in Open Promol**
1st	*y => z*	*y = x1 + x2*	***Ko*** /@**sub**[z]	@**sub**[z] = *x1 + x2*
2nd	Context =0: '+' Context =1: '*'	*y = x1 + x2*	***Ka*** /@**if**[*cnt*, {*},{+}]	*y = x1* @if[*cnt*, {*},{+}] *x2*
3rd	To summate to 10	*y = x1 + x2*	***Kc*** /@***gen***[9, {+},{x},1]	*y* =@***gen***[9, {+},{x}, 1]
Hierarchic	sum./ mult. to 10 depending on *cnt*	*y = x1 + x2*	***Ka*** imbedded in ***Kc***	*y*=@***gen***[9,{@**if**[*cnt*, {*}, {+}]},{*x*}, 1]

Note. *cnt* is a metaparameter meaning the context of use (if *cnt*=1, to perform multiplication; otherwise – summation).

__**Explanation**__:

<*semantic description for change*> serves for transferring information to metaprogrammer about semantics of change; this information is extracted from the informal description of requirements and it is used as comments in the metaprogram to be developed;

<*class of change with operation type* > may be orthogonal in three categories or hierarchical; it is obtained in step 3; it also indicates operations needed to implement a given class of changes; operations are obtained from *Table 4* (see also *step 2*); for each category, an orthogonal operation from clusters is identified separately;

<*libelled component instance*> indicates a *placeholder* within a given component instance (domain/ target program), where a given change is to be performed; to identify placeholders for change, semantic analysis is to be performed; the procedure is a part of activity of creating SM (see *step 5*);

<*change parameter*> is the metaprogram object or metaparameter through which manipulations are expressed when metaprogram is executed (simply speaking, it is a metaprogram variable to support operation declarations and their execution; it plays the same role as variable in a conventional program);

<*set of parameter values*> is a set of feasible parameter values identified during requirement analysis and semantic model formation;

<*constraints* > show forbidden combinations between parameter values.

5. To analyze a given component instance and identify placeholders within the code where changes are to be embedded; though the component consists of many parts described using multiple language (see Table 1 and model in Figure 1*c*), a metadesigner treats the component as homogeneous structure that is coded using a unique scripting language (e.g., ASP).
6. To perform a logical linking of four entities: a given component instance (see *step 5*); metaprogram structure (it consists of metainterface and metabody), clusters of a metalanguage (see Table 3) and semantic model (3). Rules for logical linking are as follows:

- __**Rule A**__: To form the metainterface of the metaprogram to be designed according to formula (4), where metainterface *MI is formally expressed through mappings and transformations of adequate sets.* Sets are found in the semantic model.

$$SM' = \left(\left(P_i\left(V_i^k\right), M\right), C\right) \overset{(K_o, K_a, K_c)\in ML}{\Rightarrow} MI\,, MI \in ML\,. \qquad (4)$$

Here *SM'* – semantic model reflecting the change aspects, where the instance is ignored (it is assumed that the instant *I* is described implicitly); P_i – (meta)parameter for change *i* (*i*=1, ... *n*); *n* – number of changes (parameters); V_i^k – set of variants for (meta)parameter *i; M* – set of modification (change) classes with identified operations, *C* – set of constraints, *MI* – metainterface*; ML* – set of constructs of a metalanguage; $\Rightarrow$ – is treated as a model transformation.

- __**Rule B**__: To codify and transform the semantic model *SM* into the metaspecification body *MB* according to formula (5), where metabody *MB is formally expressed through mappings and transformation of adequate sets*:

$$SM = \left(I(P_i; M), C\right) \overset{(K_o, K_a, K_c)\in ML}{\Rightarrow} MB\,,$$
$$MB \in \left(ML \cup TL(I)\right). \qquad (5)$$

Here *SM* is the semantic model that specifies the needed sets including those that describe changes *M*; *I* – a component instant with placeholders (see also *Table 4* and properties 3-5 in Section 5), which is seen as a function of parameters and a set of modifications *M*; *TL* – target language (languages) to describe the instant *I.*

We receive the product, the metaprogram (see (6)), as a result either of concatenation of models (4) and (5) (e.g., using Open Promol [8], where metainterface is an external entity) or integration of the models (e.g., using metalanguages, which support internal interfaces within the program structure).

$$MI \cup MB = MP\,, MP \in \left(ML \cup TL(I)\right), \qquad (6)$$

where $\bigcup$ means the integration of *MI* and *MB* according to the syntax rules of a given *ML*.

Figure 2 illustrates the result of the process of creating metaprograms according to the proposed methodology. Note that there is an illustrative example of a metaprogram only, which contains one file with metainterface (lines 3-19) and metabody (lines (22-36). This metaprogram, which is coded in ASP, generates the domain program in HTML (not shown).

```
1<%
2   ' --- Lines 3-19: metainterface in ASP language ---
3   Response.write "<form method=""POST"" action=""testas2.asp?MMPG=true&forma=true"">" & vbcrlf
4   Response.write "<p>Choose encoding <select size=""1"" name=""Language"">" & vbcrlf
5   Response.write "      <option>Baltic</option>" & vbcrlf
6   Response.write "      <option>Unicode</option>" & vbcrlf
7   Response.write "</select><br>" & vbcrlf
8   Response.write "<font size=""1""><br></font>" & vbcrlf
9   Response.write "Choose Title of web page <select size=""1"" name=""Name"">" & vbcrlf
10  Response.write "    <option>Example_1</option>" & vbcrlf
11  Response.write "    <option>Example_2</option>" & vbcrlf
12  Response.write "</select><br>" & vbcrlf
13  Response.write "<font size=""1""><br></font>" & vbcrlf
14  Response.write "Choose a content of web page <select size=""1"" name=""Body"">" & vbcrlf
15  Response.write "    <option>Test page...!</option>" & vbcrlf
16  Response.write "    <option>Hello, world!</option>" & vbcrlf
17  Response.write "</select></p>" & vbcrlf
18  Response.write "<p><input type=""submit"" value=""Generate web page"" name=""Generate""></p>" & vbcrlf
19  Response.write "</form>" & vbcrlf
20
21  ' --- Lines 22-36: specification of metabody in ASP language ---
22  If Request.QueryString ("form") = "true" then
23      Response.write "<b>Generated web page (in HTML language):</b>" & vbcrlf
24      Response.write "<xmp>" & vbcrlf
25      Response.write "<html>" & vbcrlf & "<head>" & vbcrlf
26      If Request.Form("Language") = "Baltic" Then
27          charset = "windows-1257"
28      Else
29          charset = "UTF-8"
30      End If
31      Response.write "<meta http-equiv=""Content-Type"" content=""text/html; charset=" & charset & """></head>"
32      Response.write "    <title>" & Request.Form("Name") & "</title>" & vbcrlf
33      Response.write "<body>" & vbcrlf & "    " & Request.Form("Body") & vbcrlf & "</body>" & vbcrlf
34      Response.write "</html>" & vbcrlf
35      Response.write "</xmp>"
36  End If
37%>
```

Figure 2. An illustrative example of the fragment of a metaprogram (with the metainterface and metabody in ASP)

The process of creating a metaprogram is yet not complete, if there is no evidence on quality of the product. As we use abstracts models, we ensure quality through generation and testing of generated instances. Testing should cover at least all different paths in the metaprogram. Each tested path should correspond at least to one variant for each parameter. Testing of a particular instance is similar to a conventional program testing: we need to execute the instance using the target language (interpreter/processor) and check syntactic and semantic correctness.

In this paper we provide two complexity measures to reason about the complexity of created metaprograms (see Section 8).

8. Experimental results

8.1. Analysis of characteristics of developed generators

The experimental results have been obtained through the development and the use of three Web component generators. The Generators incorporate those components that are most frequently used to implement portal-based systems (see Tables 1, 4). Using the generators, the needed component instances were automatically generated on demand depending on the context of a concrete usage. The derived instances were then integrated into 3 portal settings for different organizations. The results we describe below are split among Tables 4-6. Table 4 summarizes multi-linguistic aspects of the generators only.

Table 5 summarizes some quantitative characteristics of the developed domain generators. These characteristics identify either direct attributes or those that are further used to derive the other characteristics such as complexity measures.

Table 4. Multi-linguistic aspects of parameterized Web components described as one-stage metaprograms

No	Component generator type	Meta-language s*	Target languages	Explanation of roles of languages
1	Data management (DMG) generator	VB ASP	SQL (Server side) XML (more ser XSL (Client) HTML (Client) CSS (Client) ASP (for Scripting)	Roles of target languages were identified yet at the analysis phase (see Table 1). ASP as a target language is for scripting of other files *). ASP as a metalanguage is for specifying generation of the anticipated changes. VB as a metalanguage supports Web security aspects. Both metalanguages support generation aspects, i.e., ASP calls VB components when it is needed in the generating process and the VB part returns the generated data.
2	Content management (CMG) generator	VB ASP	XML, XSL, HTML, CSS JScript ASP	
3	Dynamic module management (DMMG) generator	ASP	XML, XSL, HTML, JScript, ASP	

*) **Note.** We distinguish two kinds of scripting: logical and physical. The first means creating links between files through the call-type statements. The second means physical composition of files resulting in the change of the file structure.

Table 5. Characteristics of metaprograms as domain generators

Generator type	Metaparameters (MP)		Constraints #	Characteristics of Internal files			Generator's (metaprogram's) size	
	MP #	Variants # of a MP [from… to]		Number of files #*)	Average # of lines/B	Total # of lines/MB	Meta-interface (lines/~MB)	Meta-body (lines/ ~MB)
DMG	14	4..16	>50**	105	106/4800 B	11130/7.3	3339/2.2	7791/5.1
CMG	21	1..50	>200**	140	210/6590 B	29400/14.4	13230/6.5	16170/7.9
DMMG	4	1..4	0	40	79/2870 B	3160/5.4	1896/3.2	1264/2.2

Notes: *) - see Figure 1, *c*, *d* and Section 5. **) Constraints are introduced through analysis: either 1) to comply standards (e.g., HTML, SQL, etc.), 2) to fulfil requirements or 3) to simplify the implementation.

8.2. Estimation of complexity of the developed generators

It is important to estimate the complexity of the developed generators (i.e., metaprograms) for many reasons (e.g., comparative study, testing and comprehension, etc.). Table 6 presents derivative characteristics of the complexity of the generators. The complexity of metaprograms can be evaluated, e.g., using such measures as LOC (lines of code), Kolmogorov's Complexity (KC), Relative Kolmogorov's Complexity (RKC) and Cyclomatic Complexity (CC). KC measures complexity of an object by the length of the smallest program that generates it. RKC is calculated as complexity of an object divided by the length of an object. A high value of RKC means that there is a high variability of metaprogram's code content, i.e., high complexity. A low value of RKC means high redundancy, i.e., the abundance of repeating code fragments in metaprogram. CC was proposed by McCabe in 1976. It directly measures the number of linearly independent paths through a program's source code from entrance to each exit. For metaprograms, CC is equal to the number of distinct domain program instances that can be generated from a metaprogram.

Table 6. Comparison of complexity measures of the developed generators and generated instances

Type	Generators' complexity				Generated instances *	
	Meta-interface (lines/~MB)	Meta-body (lines/~MB)	Kolmogorov's Complexity	Relative Kolmogorov's Complexity	Number of instances (Cyclomatic Complexity)	Average size (lines/B)
DMG	3339/2.2	7791/5.1	7805	0.157	224	120/ 4950 B
CMG	13230/6.5	16170/7.9	20123	0.101	448	240/7100 B
DMMG	1896/3.2	1264/2.2	21203	0.133	112	80/2900 B

A higher value of CC indicates higher complexity of the metaprogram's parameter set (metainterface). It is worth to know that CC depends on the number of parameters, the number of values for each parameter and the number of constraints among parameter values. As some parameters have an extremely large space of values, Table 6 represents the only lower bound on CC, i.e., the number of instances which were generated and tested.

Based on the values of these complexity metrics, we can conclude that CMG is the most complex metaprogram with the largest number of component instances that can be generated from it. However, despite its complexity, CMG still has much room for further generalization as indicated by low RKC metric language. To calculate the complexity using the Kolmogorov's complexity measure, see [10] for details.

9. Discussion and evaluation

The basic results we have achieved and described in the paper are the two: 1) the detailed process of creating externally parameterized metaprograms which are higher-level program generators for the portal domain; 2) functional characteristics and also non-functional characteristics such as complexity measures (e.g., cyclomatic number and Kolmogorov's complexity measures) of Web components generators which were used in three real portal settings. The discussion relates to those aspects that outline a specificity of the methodology used with respect to the portal domain.

Though, in general, the process of creating a metaprogram we have described in Section 7 is independent upon the application domain, nevertheless some aspects of the process may have specific features for a particular domain. For example, the Web components are described using not a unique target language but rather a set of portal-oriented languages, each expressing different aspects of that multi-faced domain. Therefore a Web component instance which is to be generalized consists of separate fragments described using different target languages. The fragments are combined in a coherent specification using ASP because of its scripting capabilities. Next, for that domain, ASP is better suited as a metalanguage for describing modifications than other metalanguages.

On the other hand, it is not enough to have only one meta-language because of the need to specify security aspects independently. Thus the metaprogramming-based specification for Web components is multi-linguistic also in terms of generalization because we need to use at least two metalanguages (e.g., VB and ASP). Furthermore, a metalanguage may perform two roles: it describes scripting and modifications in the same specification at once (as it is the case for ASP). As a result of the aforementioned features, the structure of the metaprogram that describes externally parameterized Web generators is slightly different (e.g., metainterface distributed across multiple instance fragments) in comparison to stand-alone metaprograms.

Depending on the capabilities of metalanguage, the semantic model as a basis for developing metaprograms may have a slightly different interpretation. For example, if a metalanguage supports the external interface (as it is in the case of using Open Promol), there is no need to express a component instance in the semantic model explicitly. Otherwise, if a metalanguage supports an internal interface (e.g., through the WRITE statement, see also *Table 2*) the semantic model reflects the instance explicitly. Combing these two cases into the one we can write (7):

$$SM' = \left(\left(P_i\left(V_i^k\right); M\right), C\right) \subseteq SM = \left(I\left(P_i; M\right), C\right). \quad (7)$$

Though the semantic model *SM* enables to understand the process of creating metaprograms well because of its main focus on linguistic features that are essential to form the metaprogram, however, this is not enough when the amount of changes is big and relationships (including constraints) between them are complex. The semantic model is not a weak instrument per se but rather its expressive power is not enough in that case because of complexity of the dealing problem. The complexity of the problem arises due to 1) difficulties of domain understanding and expressing its artefacts through the adequate model, 2) difficulties of implementation technology (i.e., one-stage metaprogramming with multiple metalanguages and multiple domain languages). What is the way for managing the complexity issues? We see the solution by the extending (or combining) the semantic model with feature diagrams [26], which are the more relevant models for expressing domain variability and constraints. But this is beyond the scope of the paper.

10. Conclusions and future work

The externally parameterized Web component generators that implement pre-defined and pre-programmed changes according to the principles of heterogeneous metaprogramming enable to achieve higher quality and productivity with the opportunity for wide-range adaptations in comparison to lower-level generators. But this result is due to the more extensive and accurate analysis of the portal domain per se. The foundation of applying one-stage metaprogramming techniques is a semantic model which is to be devised as a result of portal domain analysis and the analysis of requirements for change. More abstractly, parameterized Web component generators use the *white-box reuse model* combined with metaprogramming techniques. We have also identified specific features of the Web component generators in comparison to stand-alone domain program generators. Those features are: multi-linguistic aspects at two levels (meta and domain) and structural aspects of metaspecification. The complexity measures of the developed generators we present enable to evaluate the complexity of the products and reason about the limits of their evolution.

The future work will be directed at the extension of the capabilities of the semantic model with the opportunity to combine the model with feature diagrams, as well as investigation of multi-stage meta-programming techniques in the development of more powerful Web component generators.

References

[1] **K.G. Coffman, A.M. Odlyzko**. The size and growth rate of the Internet. *AT&T Labs*, 1998.

[2] **R.T. Fielding, R.N. Taylor**. Principled Design of the ModernWeb Architecture. *Proc. of* 22nd *Int. Conf. on Software Engineering* (ICSE '00), *Limerick, Ireland, June* 2000, 407–416.

[3] **T. K. Hazra.** Building Enterprise Portals: Principles to Practice. ICSE'02, 2002, *May*, 19-25, *Orlando*, 623-633.

[4] **B. Doyle, C.V. Lopes**. Survey of Technologies for Web Application Development. arXiv:0801.2618v1 [cs.SE] 2008 *January*, *http://arxiv.org/PS_cache/arxiv/pdf/0801/0801.2618v1.pdf.*

[5] **T. Helman, K. Fertalj**. A critique of Web application generators. *Proc. of the* 25*th Int. Conf. on Information Technology Interfaces*, ITI' 2003, 639 – 644.

[6] **R.B. Djemaa, I. Amous, A.B. Hamadou**. GIWA: A generator for adaptive Web applications. *Advanced Int. Conf. on Telecommunications / Int. Conf. on Internet and Web Applications and Services* (AICT/ICIW' 06), *February* 2006, *Guadeloupe, French Caribbean*,19-25.

[7] **E. Losh.** Assembly Lines: Web Generators as Hypertexts. *Proc. of the 18th Conf. on Hypertext and Hypermedia. New York: ACM Press*, 2007, 115-122.

[8] **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, Boston, November* 2003, 229-250.

[9] ASP, ASP.NET, COM, JScript, VBScript and .NET at, *http://www.microsoft.com/.*

[10] **R. Damaševičius**. On the Quantitative Estimation of Abstraction Level Increase in Metaprogramms. *Computer Science and Information Systems, Vol.*3, *No.*1, 2006, 53-64.

[11] **U. Pettersson, S. Jarzabek**. Industrial Experience with Building a Web Portal Product Line Using a Lightweight, Reactive Approach. *ESEC-FSE*'05, *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM Press, September* 2005, *Lisbon*, 326-335.

[12] **D.C. Rajapakse, S. Jarzabek**. A Need-Oriented Assessment of Technological Trends in Web Engineering. *In D. Lowe, M. Gaedke (Eds.): Proc. of* 5*th Int. Conf. on Web Engineering, ICWE* 2005, *Sydney, Australia, July* 27-29, *LNCS* 3579, *Springer*, 2005, 30-35.

[13] **R. Damaševičius, V. Štuikys.** Taxonomy of the Fundamental Concepts of Metaprogramming. *Information Technology and Control*, 37(2), 2008, 124-132.

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

[15] **S. Trujillo, M. Azanza, O. Diaz**. Generative Metaprogramming. *Proceedings of the Conf. Generative Programming and Component Engineering.* GPCE' 07, *October* 1–3, 2007, *Salzburg, Austria, ACM Publication*, 2007, 105-114.

[16] **D. da Cruz, M.J.V. Pereira, M. Béron, R. Fonseca, P.R. Henriques**. Comparing Generators for Language-based Tools. *Proc. of the 1st Conf. on Compiler Related Technologies and Applications, CoRTA*'07, *Portugal*, 2007.

[17] **K. Czarnecki, U.W. Eisenecker**. Generative Programming – Methods, Tools, and Applications. *Addison-Wesley, June* 2000.

[18] **G. Kiczales**. Aspect-Oriented Programming. *ACM Comput. Surv., Vol.*28, *No.*4, 1996.

[19] **S. Jarzabek, S. Li**. Eliminating redundancies with a "composition with adaptation" meta-programming technique. *Proc. of the* 11*th ACM SIGSOFT Symposium on Foundations of Software Engineering* 2003 *at* 9*th European Software Engineering Conference, ESEC/FSE* 2003, *Helsinki, Finland, September* 1-5, 2003, 237-246.

[20] **J. Bosch.** Design and use of software architectures: adopting and evolving a product-line approach. *ACM Press/Addison-Wesley Publishing Co., New York, NY*, 2000.

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

[22] **K. Pohl, G. Bockle, F. van der Linden.** Software Product Line Engineering. *Berlin, Heidelberg, New York: Springer-Verlag*, 2005.

[23] **T. L. Veldhuizen**. Tradeoffs of Metaprogramming. *In: ACM SIGPLAN Workshop Partial Evaluation and Semantic-Based Program Manipulation. Charleston, South Carolina* (*USA*), 2006.

[24] **T. J. Biggerstaff.** A perspective of generative reuse. *Annals of Software Engineering*, 5, 1998, 169–226.

[25] **I.D. Baxter.** Transformation Systems: Generative Reuse for Software Generation, Maintenance and Reengineering. *In C. Gacek (Ed.): Proc. of* 7*th Int. Conf. on Software Reuse: Methods, Techniques, and Tools, ICSR*-7, *Austin, TX, USA, April* 15-19, 2002, *LNCS* 2319 *Springer*, 2002, 341-342.

[26] **K.C. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson.** Feature-Oriented Domain Analysis (FODA) Feasibility Study. *SEI Technical Report CMU/SEI*-90-*TR*-021, 1990.

[27] **K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh**. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Ann. Software Eng*. 5, 1998, 143-168.

[28] **R. Prieto-Díaz**. Status Report: Software Reusability. *IEEE Software* 10(3), 1993, 61-66.

[29] **P. Grogono.** Designing for change. *In J. E. Botsford, A. Gawman, W. M. Gentleman, E. Kidd, K. A. Lyons, J. Slonim, and J. H. Johnson (Eds.): Proc. of the* 1994 *Conf. of the Centre for Advanced Studies on Collaborative Research, October* 31 – *November* 3, 1994, *Toronto, Ontario, Canada. IBM*, 1994, 21.

[30] **J. Sametinger.** Software Engineering with Reusable Components. *Springer-Verlag*, 1997.

[31] **P.J. Ashenden**. The Designer's Guide to VHDL. *Morgan Kaufmann Publishers, 2nd ed.*, 2002.

Received February 2009.