Building Process of SCT Generators

D. Radošević, I. Magdalenić, and T. Orehovački
University of Zagreb, Faculty of organization and informatics, Varaždin, Croatia
danijel.radosevic@foi.hr; ivan.magdalenic@foi.hr; tihomir.orehovacki@foi.hr

Abstract - This paper presents the building process of generators based on the SCT dynamic frames model. The SCT generator is defined by its Specification (S), Configuration (C) and Templates (T). These elements are represented graphically by Specification diagram and Configuration diagram. The building process starts by initial steps and continues by a spiral application development, based on Boehm's spiral model of software development. To evaluate the proposed building steps, an example of building SCT generator was given. The example uses the software tool developed with a purpose to implement the building steps of SCT based generators.

I. Introduction

The paper introduces a method of building code generators according to the SCT generator model [1]. The method is supported by our generator development tool, named as *Generator builder*.

Development tools for building of applications that are based on the principles of generative programming is a real need. A development tool makes it easy to build applications and prevents the appearance of typical errors in the design as described in [2] and in sintax [3]. Applications based on generative programming are special in that they usually use a domain-specific language for defining the characteristics of the application. A usage of domain specific language introduces a new level where errors can occur. To minimize errors and to help generator developers we have already created error messaging tool described in detail in [4]. Error messaging tool serves to detect typical errors that occur in phase of the source code generator creation. Defects in the development process can be avoided by following good practices and by using of an appropriate development tools. There are several already tried and tested techniques of generative design as it is shown in [5]. This paper presents a new technique of generative design that is specially adapted to SCT generator model [1]. In this paper are described some typical steps in building process of generators based on the SCT dynamic frames model. For that purpose we have developed a graphic development tool which implements this process. The process of building of the source code generator is the main step in building of generative applications. The inputs to this process are previously created prototypes of applications on the one hand and the collected knowledge about user requests and desired application features on the other. The generator developer creates Specification and the tool assists him in creation of appropriate source code templates and Configuration rules.

The remainder of the paper is structured as follows. Section 2 provides literature review of the most prominent techniques aimed for generating of programming code. Features of the SCT generator model are described in the third section. Building steps of SCT generators are explained in the fourth section. Discussion on design issues in defining generator building steps is offered in the fifth section. Section 6 gives conclusions.

II. Background to the Research

Software product lines (SPL) emerge during development of multiple versions of the same software system for different types of users. The objective of the SPL approach is to lower development expenses and concurrently increase software quality and productivity. Instead of developing each software system from the scratch, SPL approach supports reuse of product line assets such as the domain model, product line architecture, and generic components [6]. Although SPL members have many characteristics in common, they differ in certain users' requirements and implementation details called variants. The set forth variability in the SPL domain could result in a large number of possible variant combinations thus making them difficult to handle. Variability mechanisms that support automated composition and adaptation of reusable SPL assets represent an effective way of addressing the problem of managing variants. This section offers a brief overview of automated code generation techniques for handling variants in the SPL approach.

GenVoca [7] is a domain independent software development model aimed for generation of hierarchical SPL families. Fundamental features of GenVoca are realms composed of plug-compatible layers, type equations, and symmetric layers [8]. Standardized interfaces called virtual machines are a set of classes, their objects, and methods used for the implementation of SPL functionalities. Layer or component represents an implementation of particular virtual machine. If a set of layers implement the same virtual machine, then they constitute a realm or library. Each layer exports the virtual machine of the realm to which it belongs and imports interface of the realm of which parameter it contains. A particular layer is symmetric if it exports and imports the same virtual machine. Layers encapsulate transformation that maps operations and objects between export and import virtual machines. A named composition of layers used for modeling a particular software system is called a type equation. Realms and their layer define a grammar whose sentences are SPL members.

A modular framework which enables model-driven development of SPLs is called openArchitecturWare (oAW)[9]. The essential part of the oAW is a workflow engine that enables the definition of transformation workflows together with prebuilt workflow components meant for instantiation of models, validating their semantic correctness, their transformation into other models, and eventually code generation [10][11]. Workflow components are XML files which specify steps that need to be executed in a generator run. oAW has built-in support for operating on UML, UML2, EMF/Ecore, XML, Visio, and JavaBeans-based metamodels [12]. For semantic validation of models based on the set forth metamodels, oAW offers a declarative constraints checking language called Check where tests are specified as First-Order Logic formulas. After checking model constraints, valid models are combined into one model by employing model-to-model (M2M) transformations implemented using a textual language called Xtend. Result of the M2M transformation is used as a starting point for code generation which is done using an object-oriented template language called Xpand. More recently, openArchitecturWare has moved to the Eclipse Modeling Project [13].

XML-based Variant Configuration Language (XVCL) [14] is a general purpose template language based on Bassett's frame technology [15]. XVCL works on the principle of composing custom artifacts (e.g. code) from a base of generic, adaptable, and reusable domain product line assets called meta-components or x-frames. Meta-components are XML files where variation points are instrumented by XVCL commands thus facilitating automatic customization and evolution. Normalized layered hierarchy architecture of meta-components is called an x-framework. The specification frame (SPC) is the topmost x-frame which manages composition and adaptation process of a product line member. Starting with the SPC, the XVCL processor traverses an x-framework, interprets XVCL commands embedded in visited x-frames, and by conducting necessary adaptations assembles components of a specific system.

With an aim to overcome the barriers of the SPL approach adoption, Heradio et al. [16] have proposed the exemplar driven development process (EDD). The concept underlying EDD is analogy oriented development based on the similarities among domain products. EDD process starts with an implementation of domain exemplar which represents an intersection of all the product requirements within domain. Requirements which are out of the intersection scope are formally defined during the exemplar flexibilization phase. Exemplar flexibilization results in domain specific language (DSL) compiler which is used during application engineering phase for automatic generation of software products.

Similarly as XVCL, our approach is frames based. The essential difference is that SCT generator model is based on dynamic frames [1] that are created during the generation process while XVCL is based on static frames. In addition, the generator building process proposed in this paper starts from application prototype which is analogous to a development process suggested by Heradio et al. [16].

III. Generator Overview

Generators are meant for generation of code artifacts in diverse programming languages. Heretofore, they have been used in the development of web applications [17][18], web services [19][20], as well as for generation of lab based assignments [21]. The SCT generator model [1] defines the generator of source code from three core elements: Specification (S), Configuration (C) and Templates (T). Specification contains the features of the generated application in form of attribute-value pairs. Templates contain source code in a target programming language together with connections (replacing marks for insertion of variable code parts). Configuration defines the connection rules between Specification and Templates. All three model elements together constitute the SCT frame.

A particular SCT frame produces source code that could be either stored in a specific data file or included in another SCT frame. The basic idea of the generation process is shown in Figure 1. The initial SCT frame contains the initial source code template that includes connections. Source code template is file that contains source code and connections. Connections used in code templates define inclusion of content that can be from another code template, or source, if code template is omitted. Each connection has to be replaced with other source code template or value from Specification during the process of source code generation. The source code generator creates a new SCT frame for each connection. The source code of SCT frames located deeper in the hierarchy is included as the integral part of its superior SCT frame. The source code of the initial SCT frame is stored in a data file.

Since an average application contains more data files, the SCT model implies the existence of Handler. The Handler is the part of the SCT source code generator which aims to make the generator scalable in a way that it can produce more pieces of program code (e.g. program files) from the same set of Specification, Configuration and Templates. The SCT dynamic frames model enables the generation of various program units (e.g. files, classes, functions etc.) from the same Specification. Moreover, it enables the generation of different types of code e.g. JavaScript, PHP, XML, Python, Java, etc.

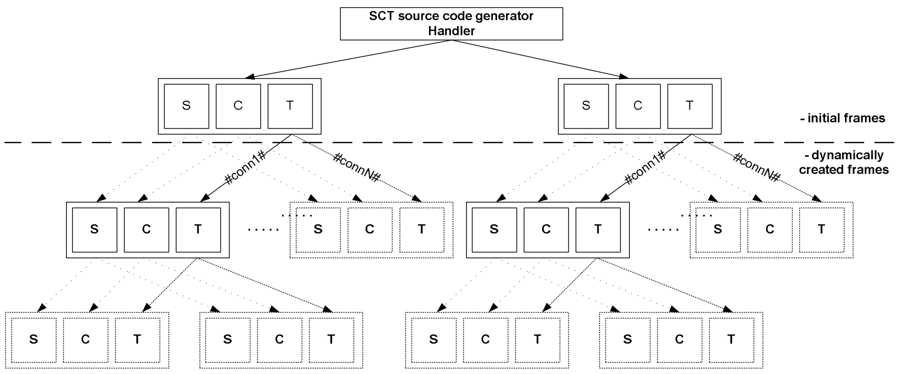

SCT source code generator Handler
S
C
T
- initial frames
- dynamically created frames
#conn1#
#connN#

Figure 1. The generation process

All three constitutive elements of SCT model are presented in case study of application which aim is dealing with dynamic list in C++. The same application is used in next section for description of building steps of a generator.

Elements of Specification are *attribute-value* pairs as shown in following example:

OUTPUT:out1
out1:output/Linked_list.cpp
field_int:student_id
field_char:surname_name
field_int:year_of_study
field_char:note

This Specification defines main features of application. The application is going to be generated in program file named *Linked_list.cpp*. Each dynamic list element consist of four attributes and their data types: *student_id* with data type *integer*, *surname_name* with data type *char*, *year_of_study* with data type *integer*, and *note* with data type *char*.

Configuration consists from Configuration rules defined by three elements: Connection, Source, and Code Template. The following example of Configuration defines 5 rules, where each rules is defined in one line.

(1) #1#,,main.template
(2) #fields_declarations#,field_*,field_*.template
(3) #data_entry#,field_*,data_entry.template
*(4) #field#,field_**
(5) #print_data#,field_*,print_data.template

The first rule defines the initial source code template *main.template*. The second rule defines replacement of connection *#fields_declarations#* with template *field_*.template* for each occurrence of attribute *field_** in Specification. For example, for attribute *field_int* is used template *field_int.template*. The third rule defines replacement of connection *#data_entry#* with template *data_entry.template* for each occurrence of attribute *field_** in Specification. Similar functionality has the fifth rule. The fourth rule defines replacement of connection *#field#* with value from specification. For example, *field_int* is replaced with *student_id* in first occurrence of this connection in source code template.

Templates are program code fragments which contain connections. Typical application is generated using several templates. For example, a template *data_entry.template* defines part of application that deals with input of user data.

cout << "#field#: ";
cin >> new_element->#field#;

The presented template has two connection *#field#* which are replaced with values from Specification. This template is used for each occurrence of attribute *field_* in Specification as defined by third rule in Configuration. The final code of that part application looks as follows:

cout << "student_id: ";
cin >> new_element->student_id;
cout << "surname_name: ";
cin >> new_element->surname_name;
cout << "year_of_study: ";
cin >> new_element->year_of_study;
cout << "note: ";
cin >> new_element->note;

All templates of example application are available online[1].

IV. BUILDING STEPS OF A GENERATOR

The process of building new generators starts from application prototype, similar to approach described in [16]. The prototype is being transformed/ decomposed into SCT model elements through several steps described in this section. The SCT generator uses these elements in automatic assembling of different application variants, which is shown on example of program in C++ that deals with simple linked list. Program variations refer to different structure of linked list element, as shown in Figure 2.

[1] SCT Generator Builder example http://gpml.foi.hr/SCT_Generator_Builder/

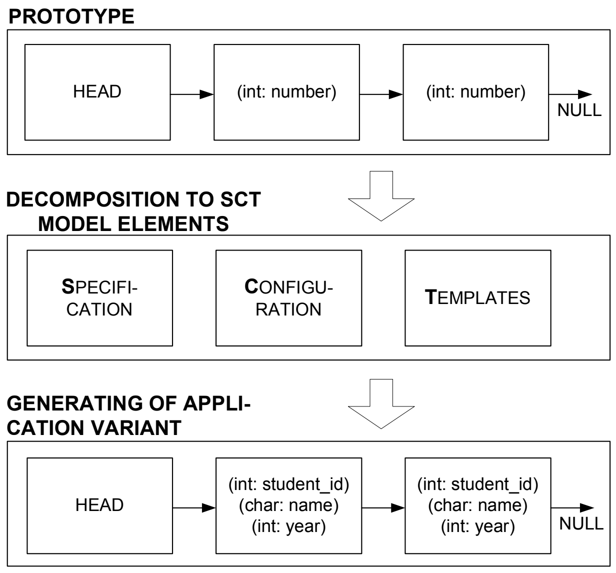

PROTOTYPE
HEAD
(int: number)
(int: number)
NULL
DECOMPOSITION TO SCT MODEL ELEMENTS
SPECIFICATION
CONFIGURATION
TEMPLATES
GENERATING OF APPLICATION VARIANT
HEAD
(int: student_id) (char: name) (int: year)
(int: student_id) (char: name) (int: year)
NULL

Figure 2. Transformation of prototype into application variant

Steps in a process of generator building, including example:

0. Prerequisite. The prerequisite for building of SCT generator is the application prototype in some textual form. This includes different types of programming code, regardless of programming language, mark-up code like HTML and XML, documentation text etc. The prototype is being relaxed through the next step in order to enable generation of variants.

1. Choosing new main templates and output types. Templates are program code fragments that contain *connections* (tags for further inclusion of code). The main templates are specified in the initial part of Configuration and define the type of code to be generated [1], e.g.:

#1#,,main.template

Each main template from Configuration is connected to corresponding output type in Specification. Output types are defined in the beginning of Specification, usually together with the names of output files to be generated, e.g.:

OUTPUT:out1	*- output type*
out1:output/program.cpp	*- output file*

2. Creating of Specification. Specification consists from *attributes* and their values. There is also a hierarchy of *attributes* that can by represent by a *Specification diagram* [1] (Figure 3).

Developer should specify attributes and their initial values for further linking to corresponding *connections* in Templates.

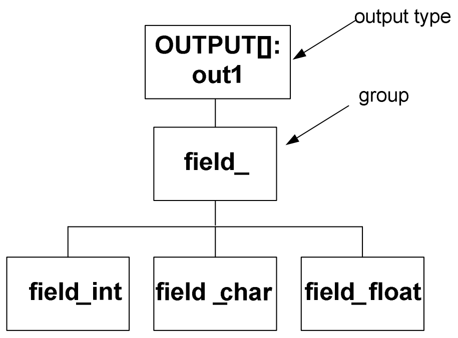


Figure 3. Specification diagram

3. Identifying of variable program parts. Variable program parts depend on Specification, so they should be later replaced with *connections*. This step is a key for *separation of concerns* [21], which is here, according to the SCT generator model, separation of program artifacts (Templates) from Specification and connection rules (Configuration). For example, the specification of used data structure contains a variable program part:

```
struct TList{
  int number;                // variable part
  TList *next;
};
```

4. Relaxing of prototype. Program parts that are identified as variable are being replaced by *connections* (in #-es). In case of data structure declaration this could look as follows:

```
struct TList{
  #fields_declarations#      // variable part
  TList *next;
};
```

5. Adding new rule to Configuration. Each *connection* created in the previous step has to be added into Configuration in form of a configuration rule. The configuration rule specifies respectively all three elements of the SCT model: connection, specification attribute and used code template. In the example this could look as follows:

#fields_declarations#,field_*,field_*.template

Specification attribute *field_** represents the number of occurrences of all attributes having name starting with *field_*. Template name specified as *field_*.template* represents usage of filenames that correspond to attribute names (e.g. usage of *field_int* designates the usage of *field_int.template*). This relation can be represented by a Configuration diagram [1] as shown in Figure 4.

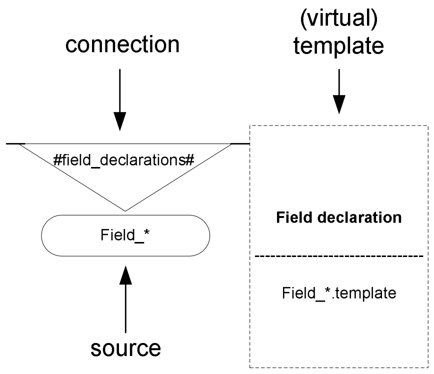


Figure 4. Part of the Configuration diagram representing single configuration rule

Usage of virtual template implies dynamic binding of template file (during the process of generation), analogous to appropriate concept of dynamic polymorphism in OOP.

6. Building of new code templates. The previous step anticipates the usage of different code templates. In case of virtual templates, there are possibly several real templates that have to be built. For a case shown in Figure 4, the templates are related to used data types, e.g.:

int #field#; *// Field_int.template*

This example requires adding of appropriate rule to Configuration (step 5):

*#field#,field_**

The third element, template, is here omitted, which specifies direct replacement of *connection* by attribute value.

7. Generating, testing and adjusting in a generative development process. The whole process of generators development and applications building consists of the repetition of operations described in the preceding steps. This can be represented by a spiral model, similar to Boehm's spiral model of software development [23], as shown in Figure 5.

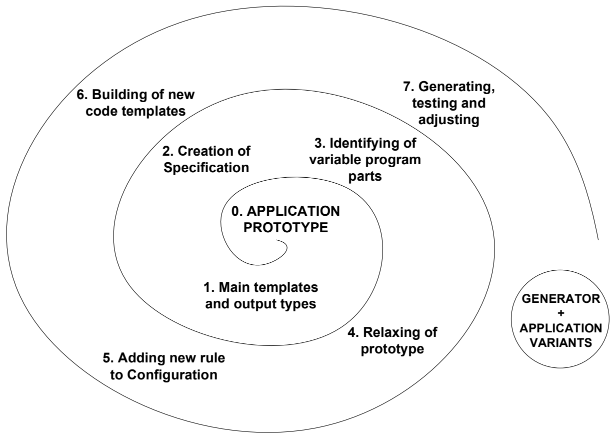
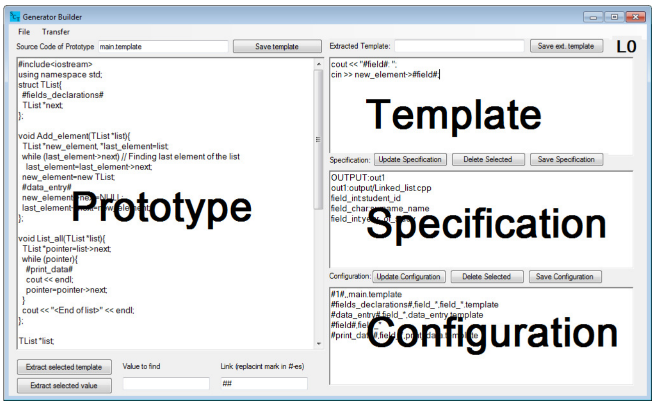

0. APPLICATION PROTOTYPE
1. Main templates and output types
2. Creation of Specification
3. Identifying of variable program parts
4. Relaxing of prototype
5. Adding new rule to Configuration
6. Building of new code templates
7. Generating, testing and adjusting
GENERATOR + APPLICATION VARIANTS

Figure 5: Spiral generator/application development

The development process starts with an application prototype and results by a generator that can be used for automatic assembling of different application variants. Modifications can start at each step of this process, following by the remaining steps, where some step could be omitted, except the step 7 (gives the final application).

V. DISCUSSION

The paper describes a method for building of code generators, according to previously introduced SCT generator model [1]. The proposed steps of generator building are also supported by a software tool named as Generator builder (Figure 6).



```
#include<iostream>
using namespace std;
struct TList{
 #fields_declarations#
 TList *next;
};

void Add_element(TList *list){
 TList *new_element, *last_element=list;
 while (last_element->next) // Finding last element of the list
  last_element=last_element->next;
 new_element=new TList;
 #data_entry#
 new_element->next=NULL;
 last_element->next=new_element;
};

void List_all(TList *list){
 TList *pointer=list->next;
 while (pointer){
  #print_data#
  cout << endl;
  pointer=pointer->next;
 }
 cout << "<End of list>" << endl;
};

TList *list;
```

```
cout << "#field#: ";
cin >> new_element->#field#;
```

```
OUTPUT:out1
out1:output/Linked_list.cpp
field_int:student_id
field_char:surname_name
field_int:year_of_study
```

```
#1#,,main.template
#fields_declarations#,field_*,field_*.template
#data_entry#,field_*,data_entry.template
#field#,field_*
#print_data#,field_*,print_data.template
```

Figure 6. Generator builder

There were some design issues in defining of generator building steps and also in defining of their order. While the starting point (*application prototype*) is rather obvious, as well as the step 1(*choosing new main templates and output types*), there were some dilemmas about the step 2. The development of a generator could start from prototype decomposition, but this approach was proved to be problematic. The problem arises from a fact that each *connection* established in a prototype (in a form of replacing marks in '#'-es) has corresponding line in Specification and also corresponding line in Configuration. Some another *connection* could be assigned to the same Specification and Configuration lines, which could lead to their repetition. To avoid repetition, it seems that it's better approach to make the Specification first. In this case, each new *connection* has to be assigned to some existing Specification line. To avoid repetition in Configuration, it's enough just to check the existence of the same rule.

Other design issues were mostly concerned to design of a software tool that could be usable in building of generators. The most important effort was to adapt the user/developer interface to the need of SCT generator model, especially in performing tasks that are not very convenient for the standard programming editors. Some of the examples are simultaneous visibility of different SCT model elements, automatic updating of Specification and Configuration, easy extracting of code templates and easy building of template variants.

There also some remaining issues like generator documenting in a form of SCT diagrams (Specification diagram and Configuration diagram) [1] and building of model elements repository, which could be some of the goals of the future work.

VI. CONCLUSION

The paper defines the development process of a SCT based code generator through seven steps. The steps are repetitive, and the whole process can be represented as a kind of spiral model of software development, similar to [23]. The appropriate software tool aimed for building of SCT based code generator, named as *Generator builder*, was also developed. For the purpose of testing the method and also the software tool, an example of generator development was given. The example starts with the prototype of a simple program in C++ that deals with linked list. The prototype was decomposed to the SCT model elements (Specification, Configuration and templates) through proposed seven steps. By using of these elements, an application variant was generated, compiled and tested.

There were some issues in design of generator building steps and software tool. Most of these were related to the order of development steps and to the design of software tool user interface. Finally, although there is a still space for improvements, the method and a software tool were found to be usable. The issues for the future work include working with the SCT diagrams and building of model elements repository.

REFERENCES

[1] D. Radošević, and I. Magdalenić, “Source Code Generator Based on Dynamic Frames”, Journal of Information and Organizational Sciences, vol. 35, no. 1, pp. 73-91, July 2011.

[2] F. Heidenreich, J. Johannes, and M. Seifert, “Generating safe template languages”, Proceedings of the 8th International ACM SIGPLAN Conference on Generative Programming and Component Engineering. Denver: ACM, 2009, pp. 99-108.

[3] J. Arnoldus, J. Bijpost, and M. Van Den Brand, “Repleo: A syntax-safe template engine”, Proceedings of the 6th International Conference on Generative Programming and Component Engineering. Salzburg: ACM, 2007, pp. 25-32.

[4] D. Radošević, I. Magdalenić, and T. Orehovački, “Error Messaging in Generative Programming”, Proceedings of the 22nd Central European Conference on Information and Intelligent Systems", Varaždin: FOI, 2011, pp. 181-187.

[5] V. Singh, and N. Gu, “Towards an integrated generative design framework”, Design Studies, vol. 33, no. 2, pp. 185‑207, March 2012.

[6] H. Zhang, and S. Jarzabek, “XVCL: a mechanism for handling variants in software product lines”, Science of Computer Programming, vol. 53, no. 3, pp. 381-407, December 2004.

[7] D. Batory, and S. O’Malley, “The Design and Implementation of Hierarchical Software Systems with Reusable Components”, ACM Transactions on Software Engineering and Methodology vol. 1, no. 4, pp. 355-398, October 1992.

[8] D. Batory, “Validating Component Compositions in Software System Generators”, Proceedings of the 4th International Conference on Software Reuse. Orlando: IEEE Press, 1996, pp. 72-81.

[9] openArchitectureWare Project, 2009, http://www.openarchitectureware.org/

[10] I. Groher, H. Papajewski, and M. Voelter, “Integrating Model-Driven Development and Software Product Line Engineering”, 2007, http://www.eclipsesummit.org/summiteurope2007/presentations/ESE2007_Model-Groher-MDDAndPLE.pdf

[11] M. Regensburger, C. Buckl, A. Knoll, and G. Schrott, “Model Based Development of Safety-Critical Systems Using Template Based Code Generation, Proceedings of the 13th IEEE International Symposium on Pacific Rim Dependable Computing. Melbourne: IEEE Press, 2007, pp. 89-92.

[12] P. Friese, and B. Kolb, “Validating Ecore models using oAW Workflow and OCL”, 2007, http://www.eclipsecon.org/summiteurope2007/presentations/ESE2007_Model-Friese-oAWAndOCL.pdf

[13] Eclipse Modeling Project, 2013, http://www.eclipse.org/modeling/

[14] T. W. Wong, S. Jarzabek, M. S. Soe, R. Shen, and H. Y. Zhang, “XML Implementation of Frame Processor,” Proceedings of the 2001 Symposium on Software reusability: putting software reuse in context ACM: Toronto, 2001, pp. 164-172.

[15] P. G. Bassett, “Framing software reuse - lessons from real world”. Toronto: Prentice Hall, 1997.

[16] R. Heradio, D. Fernandez-Amoros, L. de la Torre, and I. Abad, “Exemplar driven development of software product lines”, Expert Systems with Applications, vol. 39, no. 17, pp. 12885–12896, December 2012.

[17] D. Radošević, M. Konecki, and T. Orehovački, “Java Applications Development Based on Component and Metacomponent Approach”, Journal of Information and Organizational Sciences, vol. 32, no. 2, pp. 137-147, December 2008.

[18] D. Radošević, T. Orehovački, and M. Konecki, “PHP Scripts Generator for Remote Database Administration based on C++ Generative Objects”, Proceedings of the 30th MIPRO International Convention on Intelligent Systems. Opatija: MIPRO, 2007, pp. 167-172.

[19] I. Magdalenić, D. Radošević, and Z. Skočir, “Dynamic Generation of Web Services for Data Retrieval Using Ontology”, Informatica, vol. 20, no. 3, pp. 397-416, 2009.

[20] I. Magdalenić, D. Radošević, and T. Orehovački, “Autogenerator: Generation and execution of programming code on demand”, Expert Systems with Applications, 2013, http://dx.doi.org/10.1016/j.eswa.2012.12.003, in press.

[21] D. Radošević, T. Orehovački, and Z. Stapić, “Automatic On-line Generation of Student's Exercises in Teaching Programming”, Proceedings of the 21st Central European Conference on Information and Intelligent Systems. Varaždin: FOI, 2010, pp. 87-93.

[22] K. Czarnecki, and U. W. Eisenecker, “Generative Programming Methods, Tools, and Applications”. Boston: Addison-Wesley, 2000.

[23] B. W. Boehm, “A Spiral Model of Software Development and Enhancement”, Computer, vol. 21, no. 5, pp. 61-72, May 1988.