Towards the Software Autogeneration

D. Radosevic, T. Orehovacki and I. Magdalenic
University of Zagreb, Faculty of Organization and Informatics, Varazdin, Croatia
{danijel.radosevic, tihomir.orehovacki, ivan.magdalenic}@foi.hr

Abstract - Program generators are usually aimed for the generation of program source code. This paper introduces the idea of software source code generation and its execution on demand that we refer to as Autogeneration. Autogeneration avoids the generation of program files by using the possibility of scripting languages to evaluate program code from variables. There are several features that could be achieved by Autogeneration. Some of them are program update during its execution, optimized code without temporarily unnecessary instructions and introspection of the generation process for development purposes. An example of a web application for database content management that is implemented as an autogeneration process is presented and discussed.

I. INTRODUCTION

Program code is usually observed as a set of program files. It could be written manually or generated by usage of a software generator. Although usage of files is a usual way of storing program code scripting languages like JavaScript, Perl or Python have an alternate option, which is to evaluate program code from variables. This is a relatively scarcely researched possibility, which is mostly used for limited purposes like testing program code or generation of small program pieces.

In this paper we introduce Autogeneration, which we refer to as the automatic generation of program code and its execution on demand. In our approach, Autogeneration avoids the generation of program files in favor of using possibilities of scripting languages to evaluate program code from variables.

It is worthwhile considering possible benefits of Autogeneration. So far we have identified several possibilities enabled by such an approach. The most obvious one is the possibility to change a program specification and, consequently, its execution 'on the fly', i.e. during program execution. In addition, some program instructions, e.g. changing the database structure, are rarely used. In this case, according to our approach, the imperative statement (e.g. alter table) should be performed, and the program specification updated to the obtained new state. The code is generated in accordance with a service requested from the user, so there is no need for the regeneration of the entire application. Furthermore, there is a possibility of some introspection of the generation process. The generator can easily find the corresponding parts of the program specification, configuration and used code templates for achieving a particular user action. This could be used in the development of autogenerated software.

There are also some prerequisites and limitations of Autogeneration. Firstly, the autogenerated software has to be organized as a set of clearly distinguished services that are requested by users. This is easier to achieve in web applications, where users demand services via HTML links. Secondly, autogenerated code should be written in a scripting language (preferably the same as the generator) to work effectively. Finally, some security and performance issues need to be observed.

An example of an autogenerated web application is presented in this paper that is also available online for testing. The generator used for Autogeneration is based on our previously introduced SCT generator model [1].

The remainder of paper is organized as follows. Background to the research is described in Section 2. Basics of the SCT generation model are discussed in Section 3. An introduction to the autogeneration process is provided in Section 4. Section 5 illustrates an application example. Concluding remarks are given in the last section.

II. BACKGROUND TO THE RESEARCH

In this section we will provide a brief overview of the software development paradigms in order to create a theoretical background of the software auto-generation approach.

Software Product Line Engineering (SPLE) is a methodology of software product lines development based on the reuse of artifacts, that is, core assets. A Software Product Line (SPL) or Product Family (PF) is a group of software intensive systems sharing common set of features that meet specific needs of particular stakeholders [11]. The aim of SPLE is to reduce development time, effort, cost, and complexity, increase productivity and quality of software, and achieve higher end-user satisfaction. Rather than developing software from scratch, an existing SPL can be reconfigured and reused across projects. SPLE consists of two processes: domain engineering (in which the core assets are designed), and application engineering (in which core assets are reused during the development of a target product).

Feature Oriented Software Development (FOSD) is a common technique for representing variabilities and commonalities of a SPL. A feature is a property of a system relevant to some stakeholder used to capture

variabilities or discriminate among products in the same family [36]. Features are hierarchically organized in a diagram with a concept as a tree root. A feature model is a feature diagram that contains feature descriptions, information about stakeholders, priorities, etc. Feature modeling was proposed as a part of Feature-Oriented Domain Analysis (FODA) method for performing a domain analysis [7]. FODA provides a comprehensive description of the domain features, but neglects design and implementation phases. Therefore it has been extended to the Feature-Oriented Reuse Method (FORM), thus providing support to the object-oriented component development and architecture design [8].

Model Driven Software Development (MDSD) is a paradigm that captures essential features of a system through appropriate models [12]. In MDSD, models represent first class entities that are combined and transformed as the system is created. A modeling notation commonly referred to as Domain-Specific Language (DSL) plays the central role in MDSD [16]. DSL encompasses a meta-model that defines the abstract syntax for building models, concrete syntax description, mappings between abstract and concrete syntax, and semantics description. Former research efforts on the relationship between MDSD and SPL were mainly focused on specifying PF members by using DSLs [18]. Although MDSD has a number of benefits, a gap between specifications and their software implementations still exists [15]. With an aim to overcome the problem of having annotations scattered all over the model template, the use of Object Constraint Language (OCL) [14] notation was proposed [13].

Aspect Oriented Software Development (AOSD) aims at improving the software development process by providing modularization and composition techniques to handle crosscutting concerns [37].In general, concern is anything that is of interest to a certain stakeholder. A concern that affects multiple classes or one that is triggered in multiple situations is called a crosscutting concern. Separate modules, known as aspects, encapsulate crosscutting concerns and are subsequently composed with the rest of the system using an aspect weaver. Automatic composition of aspects with other software artifacts is either static during compilation, or dynamic at loading or runtime. There are two types of AOSD approaches. In asymmetric approaches such as AspectJ [3] there is a difference between the aspects and entities that compose the base system. Accordingly, they provide language extensions, thus declaring aspects as first class entities. On the other hand, symmetric approaches such as Hyper/J [4] assume that all concerns in a system are created equal and consequently can serve as an aspect or base in different compositions.

Frame Based Software Development (FBSD) advocates creating generalized, adapted, and thus configured components based on Frame Technology (FT). The concept of a frame as a data-structure for representing a stereotyped situations was introduced by Marvin Minsky in 1975 [19]. FT is a language independent textual pre-processor for creating systems that can be easily adapted or modified to different reuse contexts [5]. The key elements of FT are code templates organized into a hierarchy of modules known as frames, and a specification that contains particular features written by the developer. In the SPLE context, such an infrastructure embodies architectures from which SPLs are derived and evolved [6].According to an independent audit [9], FT has reduced large software project costs by over 84% and their times-to-market by 70%,concurrently reaching the reuse levels of up to 90%. The aforementioned productivity improvements motivated Jarzabek and Zhang [10] to implement XML-based Variant Configuration Language (XVCL). It is a meta-programming technique based on Basset's frames [20] to manage variabilities in SPLs. To facilitate effective reuse, XVCL enables the partitioning of programs into generic and adaptable meta-components called x-frames. An x-frame is an XML file that represents domain knowledge in the form of SPL assets. X-frames form a layered hierarchical structure called an x-framework, enabling handling variants at all granularity levels. A configuration of variants in SPL assets is recorded in a specification x-frame (SPC). Starting from the call of SPC, the XVCL Processor interprets an x-framework, performs the composition and adaptation of visited x-frames by executing XVCL commands (XML tags), and generates specific SPL members that meet specific requirements. Owing to its status as a public domain meta-language for enhancing reusability, the principles of XVCL have been thoroughly tested in practice [21][22][23].

Generative Software Development (GSD) is a widely accepted software development approach focused on the automatic generation of PF members [24]. The key concept of GSD is a generative domain model which refers to a mapping between problem space and solution space [17]. Problem space is a set of features of a PF member that are described by a DSL. On the other hand, solution space refers to implementation-based abstractions that are contained in the specification of a PF member. The mapping between the spaces is performed by means of a generator which calls a specification and results in a corresponding implementation. Apart from XVCL [10], techniques such as GenVoca [26], XFramer [25], and openArchitectureWare [27] are used for generating different types of artifacts. There are three types of generators. The ones belonging to the first type are aimed to generate code artifacts in programming languages such as PHP [28], Java [29], or Python [2]. The generators in the second type are focused on generating non-code artifacts like text [34], graphical interface [33], or students' exercises [31]. The ones in the third type are meant for building new scripting languages such as Open PROMOL [30] or CodeWorker [32], whose purpose is generator design.

Given that the afore-discussed paradigms are different but rather complementary, a number of authors (e.g. [35], [38]) have proposed the integration of two or more approaches with the aim of attaining significant synergy effects.

With a research objective of contributing to the body of knowledge on SPLE we initiated a research into the autogeneration of software. Our approach is mostly based on Generative Programming and Frame Technology with

some adjustments like the usage of dynamic frames generation [1].

III. BASICS OF THE SCT GENERATOR MODEL

The autogeneration system proposed in this paper is based on our SCT generator model [1]. For the autogeneration purpose, it is important that a generator in the base of such a system fulfills some prerequisites. Firstly, such a generator has to produce full executable program code, not only a code skeleton. Otherwise, it would not be possible to re-generate the code and execute it on each user's demand. Furthermore, the generator should be fully configurable, i.e. the configuration has to be separated from the generator code so it can be changed 'on the fly', just like in changing a program specification. Finally, the autogeneration system should use the same program specification and configuration as the 'plain' generator that generates program files. Code templates could be adapted automatically (e.g. some internal links in web applications have to be adapted for Autogeneration).

A. SCT Frame

The SCT generator model defines the source code generator on the basis of three kinds of elements [1]: Specification (S), Configuration (C) and Templates (T). All the three model elements together make the SCT frame (Fig. 1):

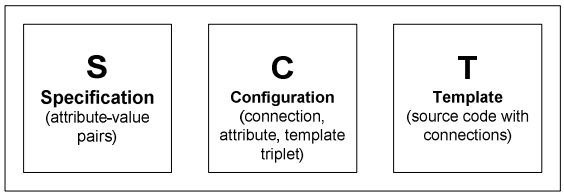

Figure 1. SCT frame [2]

Specification contains the features of a generated application in form of attribute-value pairs. Template contains the source code in a target programming language together with connections (tags for insertion of variable code parts). Configuration defines the connection rules between Specification and Templates.

B. The Generation Tree

The generation process starts from the starting, top-level SCT frame that is defined by the developer [1]. It contains the whole Specification, the whole Configuration, but only the base template from the set of all Templates. Other SCT frames are produced dynamically, during the generation process, forming the generation tree (Fig. 2).

The depth of the generation tree depends on Configuration. Configuration manages the generation process by using a set of simple rules that connect the attributes from Specification with connections (insertion tags) in Templates [1].

C. Handler

The role of Handler [2] in the original SCT model is to make the generator scalable in a way that it could produce more pieces of program code (e.g. program files) from the same set of Specification, Configuration and Templates. For the purpose of Autogeneration, Handler has been modified, as described in the next section.

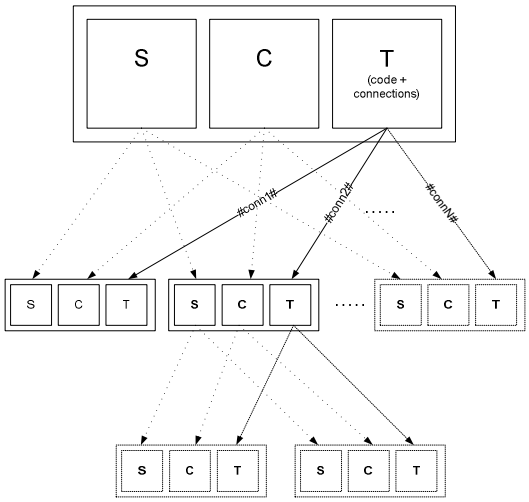

Figure 2. The generation tree [2]

IV. THE AUTOGENERATION PROCESS

The autogeneration process is described in Fig. 3. A user request is accepted by a request handler, whose task is to decompose the request and to determine what action should be taken. The request handler has to build an initial SCT frame and call the source code generator to produce the appropriate source code. It should be noted that in the autogeneration process only the source code that is needed to fulfill the user request is generated. This is the main difference in comparison with the usual use of generative programming, where the source code of a complete application is generated. This is achieved by taking a subset of Specification. Usually, the Specification contains the information needed for generating the source code of the complete application. By taking only a subset of the Specification it is possible to generate the source code needed for certain actions. Fig. 4 shows one possible subset of Specification whose purpose is to generate html templates and *cgi* scripts that deal with particular database table management.

After the request handler has built the initial SCT frame with the Specification subset, the source code generator is called to generate the program source code. The generated source code is stored in a variable, where scripting languages like JavaScript, Perl or Python can evaluate it. The Execution unit presented in Fig. 3 has the task to execute the generated source code together with arguments obtained from the request handler. Those arguments are presented in Fig. 3 as Application context and are usually obtained from user request. For example, they can include information about the user who is performing a certain action or information about a table and a record in a database that is being updated. The result of the Execution unit is sent to the user as a response to their request. In the case of a web application, the response is a web page that will be presented in the user's browser.

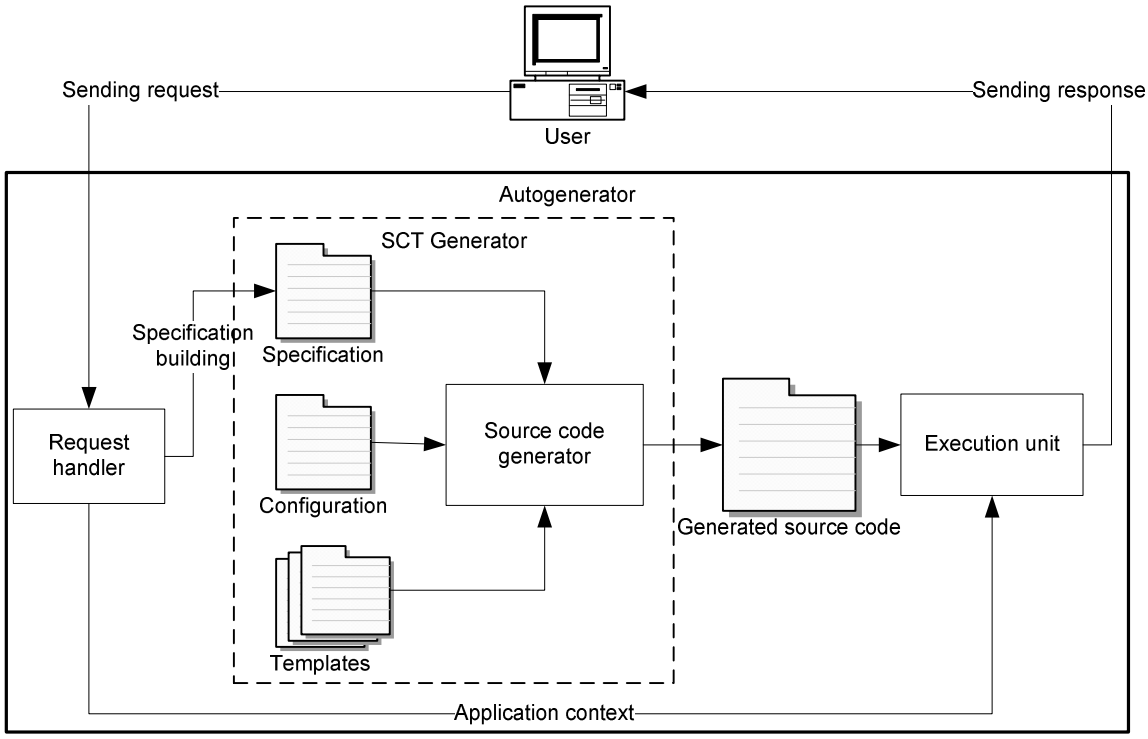

Figure 3. The autogeneration process

The autogeneration process uses two parameters (that, in case of a web autogeneration system, could be sent using the get method): the piece of code to be generated (file) and the action that should be performed (action). Specifying the file is inherited from the 'plain' generator, to maintain compatibility. As shown in Fig. 4., depending on the file, the appropriate part of Specification will be used, while depending on the action, the appropriate action will be performed (e.g. data display, data entry, data correction etc).

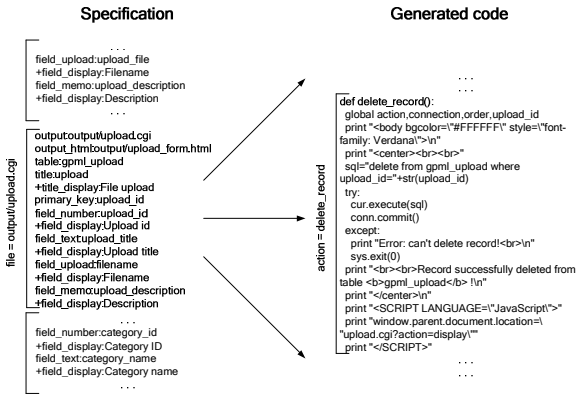

Figure 4. Specification subset

V. AN EXAMPLE

The example of the autogeneration system is a web application for database administration via web forms[1]. Basic model elements of the SCT generator model (Specification, Configuration and Templates) are available in the same way as with a conventional generator that produces program files (Fig. 5).

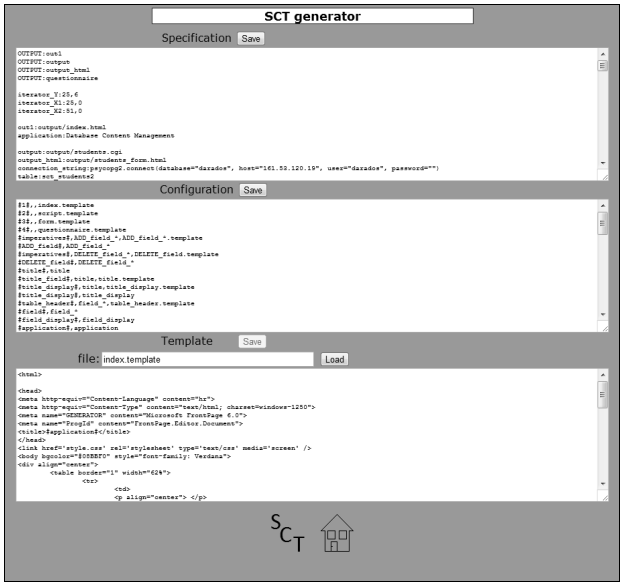

Figure 5. Basic model element in *html* form

Specification still contains some filenames, e.g.:

out1:output/index.html

The file 'output/index.html' is not generated, but the filename is used internally to denote the generated piece of code. Also, filenames are retained to keep the same specification that is used in a conventional generator.

[1]An example is available at:
http://gpml.foi.hr/SCT_Autogenerator_Example/

The generator is integrated with application execution, so that starting the generator also starts the autogenerated application (Fig. 6).

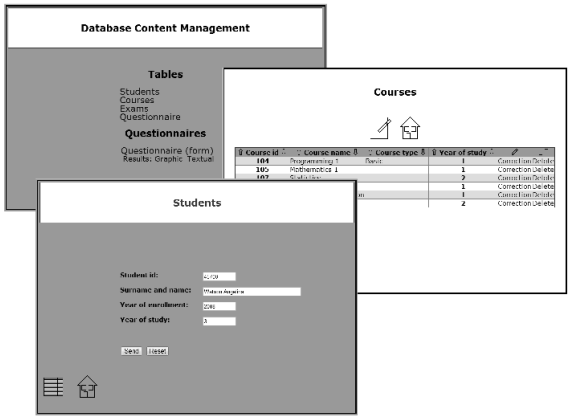

Figure 6. The autogenerated application

The example shows basic features of an autogeneration system: changing the application 'on the fly', imperative statements and code introspection.

A. *Changing the application 'on the fly'*

The code to be generated is defined by a parameter file (submitted via the get method). Any change in Specification is updated each time the user requests the re-generation and execution of the appropriatepiece of code. Configuration and Templates can also be modified 'on the fly', enabling even a substantial change in the application structure.

B. *Imperative statements*

Imperative statements in Specification are used to perform rarely used instructions, usually connected with some program dependencies, like databases. A typical example of usage of such statements is changing a database table structure, along with the change in program code. For example, the Specification statement:

ADD_field_int:new_column

will cause the generation of the appropriate ALTER TABLE statement in the generated code. After the instruction is executed, the specification will be updated by removing the imperative statement (here: ADD):

field_int:new_column

So, the imperative statement is intended to be performed once, establishing a new state.

C. *Code introspection*

Introspection in an autogeneration system enables application developers to see exactly which part of Specification, Configuration and Templates was used in the generation of acurrently executing part of an application. In the example application, introspection is implemented in form of an introspection pane, as shown in Fig. 7.

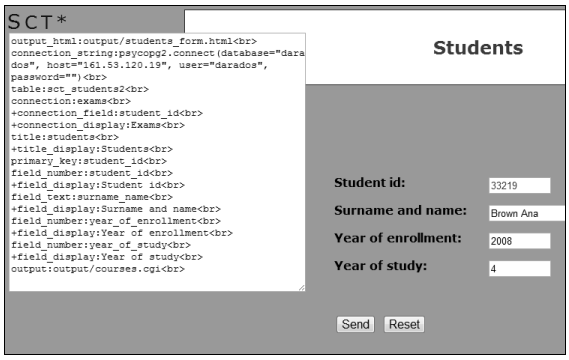

Figure 7. The introspection pane

The introspection starts from the generator' 'knowledge' of how something was generated, helping developers to find possible errors or possibilities for application improvement.

VI. CONCLUSION

The paper introduces the idea of software source code generation and its execution on demand that we refer to as Autogeneration. The presented example of a web application that works as an autogeneration system shows that such a concept is possible and could have some advantages in comparison with the usual way of code generation (into program files). Some of the possible benefits of such an autogeneration system include changing the application 'on the fly', imperative statements and code introspection. All these three concepts are included in the example application.

In addition to the benefits of using Autogeneration, there are some limitations – and even disadvantages – of such a concept. Firstly, the concept is closely connected to scripting languages (we used Python) that contain the possibility of code evaluation from variables. Furthermore, the autogenerated application also needs to be in a scripting language, preferably the same one as the generator.

The performances of such systems could therefore be slightly degraded in relation to applications generated/written in a usual way.

In our future work, we plan to define a formal model of Autogeneration and test the concept in the development of different kinds of applications.

REFERENCES

[1] D. Radošević, and I. Magdalenić, "Source Code Generator Based on Dynamic Frames",Journal of Information and Organizational Sciences, vol. 35, no. 2, pp. 73–91, 2011.

[2] D. Radošević, and I. Magdalenić, "Python Implementation of Source Code Generator Based on Dynamic Frames", In Proceedings of the 34th MIPRO International Convention, N. Bogunović and S. Ribarić, Eds. Opatija: MIPRO, 2011, pp. 369–374.

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "Getting started with AspectJ", Communications of the ACM, vol. 44, no. 10, pp. 59–65, 2001.

[4] A. Lai, G.C. Murphy, and R.J. Walker, "Separating concerns with Hyper/J: An Experience Report", In Workshop on Multi-Dimensional Separation of Concerns in Software Engineering,

http://www.research.ibm.com/hyperspace/workshops/icse2000/Papers/lai.pdf

[5] N. Loughran, A. Rashid, W. Zhang, and S. Jarzabek, "Supporting Product Line Evolution with Framed Aspects", InWorkshop on Aspects, Components and Patterns for Infrastructure Software, http://www.comp.lancs.ac.uk/computing/aop/papers/SPL_ACP4IS 2004.pdf

[6] P.G. Bassett, "The Case for Frame-Based Software Engineering", IEEE Software, vol. 24, no. 4, pp. 90-99, 2007.

[7] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report, Software Engineering Institute, Carnegie Mellon University, 1990.

[8] K. C. Kang, J. Lee, and P. Donohoe, "Feature-Oriented Product Line Engineering", IEEE Software, vol. 19, no. 4, pp. 58-65, 2002.

[9] I. Grossman and M. Mah, "Independent Research Study of Software Reuse Using Frame Technology", Technical Report, QSM Associates, 1994.

[10] S. Jarzabek, and H. Zhang, "XML-based Method and Toolfor Handling Variant Requirements in Domain Models", In Proceedings of the Fifth IEEE International Symposium on Requirements Engineering. Toronto: IEEE Press, 2001. pp. 166-173.

[11] P. Clements, and L. Northrop, "Software Product Lines: Practices and Patterns". Boston: Addison-Wesley, 2002.

[12] T. Stahl, and M. Völter, "Model-Driven Software Development:Technology, Engineering, Management". West Sussex: Wiley & Sons, 2006.

[13] K. Czarnecki, C. H. P. Kim, and K. T. Kalleberg, "Feature models are views on ontologies", In Proceedings of the 10th International Software Product Lines Conference. Baltimore: IEEE Press, 2006,pp. 41-51.

[14] J. B. Warmer, and A. G. Kleppe, "The Object Constraint Language: Precise Modeling With UML". Boston: Addison-Wesley, 1998.

[15] V. V. G. Neto, and J. L. de Oliveira, "An early aspect for model-driven transformers engineering", In Proceedings of the 2011 International Workshop on Early Aspects. Porto de Galinhas: ACM, 2011. pp. 7-11.

[16] M. Fowler, "Domain-Specific Languages". Boston: Addison-Wesley, 2010.

[17] K. Czarnecki, "Overview of generative software development", Lecture Notes in Computer Science, vol. 3566, pp. 326-341, 2005.

[18] J. Greenfield, and K. Short, "Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools". Indiana: Wiley Publishing, 2004.

[19] M. Minsky, "A framework for representing knowledge", The Psychology of Computer Vision, P. Winston, Ed., New York: McGraw-Hill, 1975.

[20] P. G. Bassett, "Framing software reuse - lessons from real world". Prentice Hall, 1997.

[21] H. Zhang, andS. Jarzabek, "XVCL: a mechanism for handling variants in software product lines", Science of Computer Programming, vol. 53, no. 3, pp. 381-407, 2004.

[22] L. Yuan, J. Song Dong, and J. Sun, "Modeling and Customization of Fault Tolerant Architecture using Object-Z/XVCL", In Proceedings of the 13th Asia Pacific Software Engineering Conference. Kanpur: IEEE Press, 2006. pp. 209-216.

[23] S. Guo, L. Tang, and W. Xu, "XVCL-An Annotative Approach to Feature-Oriented Programming", In Proceedings of the 2010 International Conference on Computational Intelligence and Software Engineering. Wuhan: IEEE Press, 2010. pp. 1-5.

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

[25] M. Emrich, and M. Schlee, "Codegenerierung mit XFramer und Programmiertechniken für Frames", Objektspektrum, no. 5, pp. 55-60, 2003.

[26] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin, "The GenVoca model of software-system generators", IEEE Software, vol. 11, no. 5, pp. 89-94, 1994.

[27] A. Haase, M. Völter, S. Efftinge, and B. Kolb, "Introduction to openArchitectureWare 4.1.2", 2007.

http://www.voelter.de/data/workshops/oawSubmissionToolsWorkshop.pdf

[28] D. Radošević, T. Orehovački, and M. Konecki, "PHP Scripts Generator for Remote Database Administration based on C++ Generative Objects". In Proceedings of the 30th MIPRO International Convention on Intelligent Systems, Opatija: MIPRO, 2007. pp. 167–171.

[29] 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, 2008.

[30] V. Štuikys, and R. Damaševičius, "Scripting Language Open PROMOL and its Processor", Informatica, vol. 11, no. 1, pp, 71-86, 2000.

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

[32] C. Lemaire, "CodeWorker Parsing tool and Code generator - User's guide & Reference manual, Release 4.5.4, 2010, http://www.codeworker.org/CodeWorker.pdf

[33] M. Schlee, and J. Vanderdonckt, "Generative Programming of Graphical User Interfaces", In Proceedings of the Working Conference on Advanced Visual Interfaces. Gallipoli: ACM, 2004. pp. 403-406.

[34] J. Műller, and U. W. Eisenecker, "The Applicability of Common Generative Techniques for Textual Non-Code Artifact Generation", In Proceedings of the Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering, Department of Informatics and Mathematics, University of Passau, 2008. http://www.infosun.fim.uni-passau.de/spl/apel/McGPLE2008/papers/Paper8.pdf

[35] I. Groher, and M. Voelter, "Aspect-Oriented Model-Driven Software Product Line Engineering", Lecture Notes in Computer Science, vol. 5560, pp. 111-152, 2009.

[36] A. Classen, P. Heymans, and P-Y. Schobbens, "What's in a Feature: A Requirements Engineering Perspective", Lecture Notes in Computer Science, vol. 4961, pp. 16-30, 2008.

[37] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming", Lecture Notes in Computer Science, vol. 1241, pp. 220-242, 1997.

[38] L. Fuentes, C. Nebrera, and P. Sánchez, "Feature-Oriented Model-Driven Software Product Lines: The TENTE approach", In Proceedings of the Forum of the 21st International Conference on Advanced Information Systems (CAiSE), E. Yu, J. Eder, and C. Rolland, Eds. Amsterdam, 2009. pp. 67-72.