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 generation of program source code. This paper introduces the idea of software source code generation and its execution on demand that we named as Autogeneration. Autogeneration avoids 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 generation process for development purposes. An example of web application for database content management that is generated according to a service requested from user, and program specification to obtained new state. The code is generated according to a service requested from user, and there is no need for 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 program specification, configuration and used code templates for achieving particular user action. This could be used in development of autogenerated software.

I. INTRODUCTION

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

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

What would be the possible benefits of Autogeneration? So far we have found several possibilities to be achieved. The most obvious one is possibility to change program specification and, consequently, its execution "on the fly" - during program execution. Then, some program instructions are rarely used, e.g. changing database structure. In this case, according to our approach, the imperative statement (e.g. alter table) should be performed, along with updating of program specification to obtained new state. The code is generated according to a service requested from user, and there is no need for 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 program specification, configuration and used code templates for achieving particular user action. This could be used in development of autogenerated software.

There also some prerequisites and limitations of Autogeneration. At first, the autogenerated software has to be organized as a set of clearly distinguished services that are requested by users. It's easier to achieve this at web applications, where users demand services via HTML links. Then, autogenerated code should be in some scripting language (preferably same as the generator) to work effectively. Finally, some security and performances issues should be observed.

An example of autogenerated web application is given in this paper, and 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 the 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 thus creating the theoretical background of the software auto-generation approach.

Software Product Line Engineering (SPLE) is a methodology of the 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 SPL 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. 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 into the Feature-Oriented Reuse Method (FORM) thus providing the 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. The central role in MDSD plays modeling notation often called Domain-Specific Language (DSL) [16]. DSL encompasses a meta-model that defines the abstract syntax for building models, concrete syntax description, mappings between abstract and concrete syntax, and a description of the semantics. Former research effort on the relationship between MDSD and SPL was 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 have been proposed [13].

Aspect Oriented Software Development (AOSD) aims at improving software development process by providing modularization and composition techniques to handle crosscutting concerns [37]. Concern is anything that is of interest to some stakeholder whereas concern that affects multiple classes or 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 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. Asymmetric approaches such as AspectJ [3] consider that 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 HyperJ [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 developer. In SPL, context, such an infrastructure embodies architectures from which SPLs are derived and evolved [6]. According to the 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 up to 90%. The set forth productivity improvements have 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 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 thus 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 composition and adaptation of visited x-frames by executing XVCL commands (XML tags), and generates specific SPL members that meet specific requirements. Being 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 widely accepted software development approach focused on automatic generation of PF members [24]. The key concept of GSD is generative domain model which refers to a mapping between problem space and solution space [17]. Problem space is a set of the features of a PF member that are described by a DSL. On the other hand, the 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 generator which calls a specification and results with the 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 first ones are aimed to generate code artifacts in programming languages such as PHP [28], Java [29], or Python [2]. The second ones are focused on generating non-code artifacts like text [34], graphical interface [33], or student’s exercises [31]. The last ones are meant for building new scripting languages such as Open PROMOL [30] or CodeWorker [32] whose purpose is the design of generators.

Given that 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 aiming to attain the significant synergy effects.

With a research objective of contributing to the SPL community of knowledge we have initiated a research into the autogeneration of software. Our approach is mostly based on Generative Programming and Frame Technology with some adjustments, like 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 autogeneration purpose, it is important that generator in the base of such system fulfills some prerequisites. At first, such generator
has to produce full executable program code, not just some kind of skeleton. In the opposite, it could not be possible to re-generate code and execute it for each user demand. Furthermore, generator should be fully configurable, i.e. configuration has to be separated from generator code in order to change it "on the fly", just like changing program specification. Finally, the autogeneration system should use the same program specification and configuration as "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 from three kinds of elements [1]: Specification (S), Configuration (C) and Templates (T). All three model elements together make the SCT frame (Fig. 1):

![Figure 1. SCT frame [2]](image)


B. The Generation Tree

The generation process starts from starting, top-level SCT frame that is defined by 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).

![Figure 2. The generation tree [2]](image)

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

C. Handler

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

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 to take. The request handler has to build initial SCT frame and call source code generator to produce appropriate source code. It should be noted that in the autogeneration process is generated only source code that is needed to fulfill the user request. This is the main difference in comparison with usual use of generative programming where the source code of a complete application is generated. This is achieved by taking a subset of a Specification. Usually, the Specification contains information needed for generating the source code of complete application. By taking only a subset of the Specification is possible to generate the source code needed for certain actions. In Fig. 4 is presented 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 initial SCT frame with Specification subset, the source code generator is called to generate program source code. The generated source code is stored in variable where scripting languages like JavaScript, Perl or Python can evaluate it. The Execution unit presented in Fig. 3 has the task to execute generated source code together with arguments given from the request handler. Those arguments are presented in Fig. 3 as Application context and usually are obtained from user request. For example, that can be information about the user who is performing certain action or information about a table and a record in a database that is being updated. The result of Execution unit is sent to user as a response to his request. In the case of web application, the response is a web page that will be presented in the user browser.

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

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

Specification still contains some filenames, e.g.:

```
out1:output/index.html
```

The file "output/index.html" is not generated, but filename is being used internally, to signify generated piece of code. Also, filenames are retained to keep same specification that is used at "regular" generator.

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

---

1 An example is available at:
http://gpml.foi.hri/SCT_Autogenerator_Example/
The example shows basic features of an autogeneration system: changing application "on the fly", imperative statements and code introspection.

A. Changing application “on the fly”

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

B. Imperative statements

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

ADD_field_int:new_column

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

field_int:new_column

So, 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 generation of, currently executing, part of application. In the example application, introspection is implemented in form of introspection pane, Fig. 7.

The introspection starts from generator's "knowing" 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 named as Autogeneration. Given example of web application that works as an autogeneration system shows that such concept is possible, and could have some advantages in relation to usual way of code generation (into program files). Some of the possible benefits of such autogeneration system could be changing 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 concept. Firstly, the concept is closely connected to scripting languages (we used Python) that have the possibility of code evaluation from variables. Then, the autogenerated application should also be in scripting language, preferably the same one as the generator.

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

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

REFERENCES


