Expert Systems with Applications 40 (2013) 2845–2857

Contents lists available at SciVerse ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

# Autogenerator: Generation and execution of programming code on demand

Ivan Magdalenić *, Danijel Radošević, Tihomir Orehovački

*University of Zagreb, Faculty of Organization and Informatics, Pavlinska 2, 42 000 Varaždin, Croatia*

ARTICLE INFO

*Keywords:*
Autogenerator
Dynamic frames
Generation on demand

ABSTRACT

While generating program files that can be executed afterwards is widely established in Automatic programming, the generation of programming code and its execution on demand without creating program files is still a challenge. In the approach presented in this paper a generator entitled *Autogenerator* uses the ability of scripting languages to evaluate programming code from a variable. The main benefits of this approach lie in facilitating the application change during its execution on the one hand and in dependencies update on the other. Autogenerator can be useful in the development of a common Generative programming application for previewing the application before the generation of the final release. With the aim of examining specific facets of the autogeneration process, we also conducted performance tests. Finally, the presented model of Autogenerator is verified through the development of an application for the creation and handling of Universal Business Language documents.

© 2012 Elsevier Ltd. All rights reserved.

## 1. Introduction

The main aspiration of Generative programming as a discipline within Automatic programming is to use generators to facilitate the process of application development. Generators are commonly used to generate entire applications or their parts for their later execution. However, the generation of program code and its execution on demand still presents a challenge. The generation of program code placed in variables and its immediate execution on user's demand are two key properties underlying Autogenerator. The program code is generated from code templates and a program specification, in accordance with configuration rules. In this paper we present a model of generation of program code and its execution on demand on the one hand and its strengths and weaknesses, as well as its possible application, on the other. Although the concept itself may be useful in the development of generators other benefits of such an approach are also highlighted. The first and most obvious feature of Autogenerator is its possibility to change the application 'on the fly' owing to the avoidance of program files. Any change in the definition of an application is updated instantly during its execution without the need of restarting the application. Another advantage of Autogenerator is the ease of handling software dependencies by means of imperative statements in the specification. Such statements enable for some new state (e.g. database table structure) to be reached and for the specification to be updated accordingly. Finally, Autogenerator supports introspection, i.e. at any point of application execution the part of the specification being used as well as its related configuration and the set of code templates are visible for the programmer.

Autogenerator is convenient for areas where time is not a crucial factor since the execution time with regards to the previously generated code is longer due to internal generation of program code on each request. Considering that the program code is first stored into variables and then executed, the programming language used needs to support this feature. Scripting languages like Perl and Python are adequate for this purpose since they function as interpreters and have the option of dynamic evaluation of program code positioned in variables. This possibility has so far been relatively rarely used for large amounts of code. Finally, applications most suitable for Autogenerator are the ones that work on the request–respond principle. This particularly applies to web applications, since each operation can be specified by a set of parameters (using the GET or POST method). Such a set of parameters also designates the required pieces of the program code to be generated and executed immediately, so there is no need to regenerate the whole application on each user request. The source code generator at the heart of Autogenerator is based on our previously introduced dynamic frames model, named SCT after the model elements: Specification, Configuration and Templates (Radošević & Magdalenić, 2011b). The great flexibility of the SCT model enables the implementation of Autogenerator that is hard to achieve by other approaches like XVCL (Jarzabek, Bassett, Zhang, & Zhang, 2003) or CodeWorker (Lemaire, 2010). The SCT model defines the generator from a set of artefacts that are specially developed for this purpose. In our case, artefacts refer to code templates (entitled *Templates*, i.e. set of all code templates), and application features (entitled *Specification*) that are synthesized according to connection rules (entitled *Configuration*). All the three elements of the

* Corresponding author. Tel.: +385 42 390 872; fax: +385 42 213 413.
*E-mail address:* ivan.magdalenic@foi.hr (I. Magdalenić).

0957-4174/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.eswa.2012.12.003

model, i.e. Specification, Configuration and Templates (SCT) define a top-level generator frame. During the generation process, new SCT frames are derived from the initial one, forming a multi-level tree structure. This allows for the nesting of generators, similar to nesting of program structures in structured and object-oriented programming. The fact that the model enables specifying and generating of more programming units from the same Specification may be used in the generation of web applications, which usually consist of a number of program parts written in different languages (e.g. HTML, PHP, Javascript, etc.).

When the implementation of Autogenerator is concerned, it is important to note that the SCT model is aimed for developing entire applications rather than skeletons that need to be subsequently completed. The implementation language of our Autogenerator prototype is Python, with the generated code to be executed from variables also written in that language. Python is used due to its flexibility as a scripting language, including the possibility of code evaluation from a variable and its object-oriented features. While some other scripting languages such as Perl and PHP are well suited only for character strings processing, Python lends itself to manipulating complex classes as well. Furthermore, Python uses flexible data structures like lists and dictionaries that contain elements of different types. Any element of such structures can be easily replaced by another, regardless of their (possibly incompatible) types, which is useful in code generation (Radošević & Magdalenić, 2011a). In our paper, the practical applicability and verification of the proposed model of Autogenerator is presented on an example of an application developed for creation and handling of Universal Business Language (UBL) documents. UBL is an e-business standard which consists of 32 e-business documents, each of them containing several thousand unique elements (OASIS, 2006). Building of input forms for such documents is a challenging task from both the designer and the programmer's point of view. The customization of these documents to an individual user is demanding since each user typically requires a subset of a large number of elements. Although this problem can be solved with parameterization, the use of Autogenerator appears to be a more natural solution. We therefore explored the problem of performance degradation that results from the usage of Autogenerator in relation to a conventional web application in order to identify the main factors that affect it. For that purpose, the application in form of Autogenerator was compared with a previously generated application of the same functionality. The response time of the two analogous web applications was measured, as well as the time needed for the generation of necessary pieces of code.

The paper is organized as follows: Section 2 provides the background to our research; Section 3 shortly describes the SCT model that Autogenerator is based on; the Autogenerator model is presented in Section 4; an example of Autogenerator and its features are demonstrated in Section 5; Section 6 contains arguments concerning performance testing results; the final section offers concluding remarks and guidelines for future work.

## 2. Background to research

The development of Autogenerator was influenced by several software development paradigms. This section provides their brief overview, which constitutes the theoretical background to the software autogeneration approach.

Software Product Line Engineering (SPLE) is a methodology of software product lines development based on the reuse of artefacts, i.e., core assets. A Software Product Line (SPL) or Product Family (PF) is a group of software intensive systems that share a common set of features that meet particular stakeholders' specific needs (Clements & Northrop, 2002). Among the aims of SPLE are the reduction of development time, effort, cost, and complexity on the one hand and the increase in productivity and quality of software and higher end-user satisfaction on the other. The fact that an existing SPL can be reconfigured and reused across different projects means that developing software from scratch can be avoided. Two processes are comprised in SPLE: 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 commonly used technique for representing variability and commonalities of a SPL. The term feature refers to a property of a system relevant to some stakeholder that is used to capture variability or discriminate among products in the same family (Classen, Heymans, & Schobbens, 2008). 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 modelling was proposed within the Feature-Oriented Domain Analysis (FODA) method for performing domain analysis (Kang, Cohen, Hess, Novak, & Peterson, 1990). While providing a comprehensive description of domain features, FODA neglects design and implementation phases. To address this issue, its extension into the Feature-Oriented Reuse Method (FORM) was devised, providing support to object-oriented component development and architecture design (Kang, Lee, & Donohoe, 2002). In FOSD software features are treated as fundamental units of abstraction and composition. The approach proposed in this paper is similar to FOSD since features of Autogenerator are contained in Specification which defines one application from a set of possible applications. Code templates, which act as main building blocks, contain connections that can be used for adding different crosscutting concerns. Finally, we use the Configuration of the source code generator to perform problem-domain adjustments by means of pre-processor definitions, as suggested by Rosenmüller, Siegmund, Saake, and Apel (2008). The SCT model is oriented to working with code-fragment-sized components, following the approach in XVCL (Jarzabek et al., 2003). In other GP based projects like Uniframe (Olson, Raje, Bryant, Burt, & Auguston, 2005) descending to code-fragment-sized components is avoided. Since our components are not necessarily strictly connected to program organizational units such as classes or methods, our approach differs from the metaclass-based approaches described by Grigorenko, Saabas, and Tyugu (2005), Tolvanen and Rossi (2003) and De Lara and Vangheluwe (2004).

Model Driven Software Development (MDSD) is a paradigm in which essential features of a system are captured through appropriate models (Stahl & Völter, 2006). In MDSD, models represent first class entities that are combined and transformed in the process of system creation. Modelling notation commonly termed Domain-Specific Language (DSL) (Fowler, 2010) plays a central role in MDSD. DSL comprises a meta-model by means of which the abstract syntax for building models, concrete syntax description including mappings between the abstract and concrete syntax are defined, along with the description of semantics. Former research into the relationship between MDSD and SPL was primarily focused on specifying PF members by using DSLs (Greenfield & Short, 2004). Although a number of benefits of MDSD have been established, the connection between specifications and their software implementations has not been sufficiently explored (Neto & de Oliveira, 2011). In order to address the issue of having annotations scattered all over the model template, the use of Object Constraint Language (OCL) (Warmer & Kleppe, 1998) notation was proposed (Czarnecki, Kim, & Kalleberg, 2006). The behaviour of Autogenerator is defined by Configuration, where instructions for building the source code are stored. Both textual notation (explained in Section 3), and XML notation (Radošević & Magdalenić, 2011b) can be used for that purpose.

Aspect Oriented Software Development (AOSD) is aimed at improving the software development process by providing modularization and composition techniques to handle crosscutting concerns (Kiczales et al., 1997). While the term concern generally refers to anything that is of interest to a stakeholder, a concern that affects multiple classes or the one that is triggered in multiple situations is called a crosscutting concern (Kiczales et al., 1997). Such crosscutting concerns are encapsulated in separate modules, known as aspects, and are subsequently composed with the rest of the system using an aspect weaver. Automatic composition of aspects with other software artefacts is either static during compilation, or dynamic at loading or runtime. Two types of AOSD approaches exist. In asymmetric approaches such as AspectJ (Kiczales et al., 2001) it is assumed that there is a difference between aspects and entities that compose the base system. Accordingly, they provide language extensions whereby aspects are declared as first class entities. On the other hand, in symmetric approaches such as Hyper/J (Lai, Murphy, & Walker, 2000) it is assumed that all concerns in a system are created equal and can consequently act as an aspect or base in different compositions. Since Autogenerator's templates offer such functionality, they can be treated as crosscutting concerns.

In Frame Based Software Development (FBSD) the creation of generalized, adapted, and thus configured components based on Frame Technology (FT) is advocated. The concept of a frame as a data-structure for representing stereotyped situations was introduced by Minsky (1975). FT is a language independent textual pre-processor for creating systems that can be easily adapted or modified to different reuse contexts (Loughran, Rashid, Zhang, & Jarzabek, 2004). The distinguishing 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 (Bassett, 2007). In an independent audit (Grossman & Mah, 1994) it was established that FT reduced large software project costs by over 84% and their times to-market by 70%, while reaching the reuse levels of up to 90%. Following the afore-mentioned productivity improvements, Jarzabek and Zhang (2001) implemented the XML-based Variant Configuration Language (XVCL), which is a meta-programming technique based on Basset's frames (Bassett, 1997) to manage variability in SPLs. XVCL enables partitioning of programs into generic and adaptable meta-components called x-frames, thus facilitating effective reuse. An x-frame is an XML file that represents domain knowledge in form of SPL assets. X-frames form a layered hierarchical structure called an x-framework that enables handling variants at all granularity levels. A configuration of variants in SPL assets is recorded in a specification x-frame (SPC). Starting from the SPC call, 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. Owing to its status as a public domain meta-language for reusability enhancement, the principles of XVCL have been thoroughly tested in practice (Guo, Tang, & Xu, 2010; Yuan, Song Dong, & Sun, 2006; Zhang & Jarzabek, 2004). Autogenerator is based on the SCT model that represents an example of advanced use of FBSD. Unlike XVCL, Autogenerator relies on dynamic frames, which means that frames are dynamically created during the source code generation process (Radošević & Magdalenić, 2011b; Radošević, Orehovački, & Magdalenić, 2012). This increases flexibility in the development of generators with regards to the use of static frames.

Generative Software Development (GSD) is a widely accepted software development approach that relies on automatic generation of PF members (Czarnecki & Eisenecker, 2000). The key concept behind GSD is a generative domain model that focuses on mapping between problem space and solution space (Czarnecki, 2005). While problem space refers to a set of the features of a PF member that are described by a DSL, solution space denotes implementation-based abstractions that are contained in the specification of a PF member. The mapping between these two spaces is performed by means of a generator which calls a specification and yields a corresponding implementation. In addition to XVCL (Jarzabek & Zhang, 2001), other techniques including GenVoca (Batory et al., 1994), XFramer (Emrich & Schlee, 2003), and openArchitectureWare (Haase, Völter, Efftinge, & Kolb, 2007) are used for generating different types of artefacts. Several types of generators exist. Whereas some generators are aimed to generate code artefacts in programming languages such as PHP (Radošević, Orehovački, & Konecki, 2007), Java (Radošević, Konecki, & Orehovački, 2008), or Python (Radošević & Magdalenić, 2011a), other generators are mainly used for creating non-code artefacts that range from text (Műller & Eisenecker, 2008) and graphical interface (Schlee & Vanderdonckt, 2004) to student assignments (Radošević, Orehovački, & Stapić, 2010). Some built in generators can be found in specialized programming languages such as Open PROMOL (Štuikys & Damaševičius, 2000) or CodeWorker (Lemaire, 2010) designed for generators development. However, in some cases there is no alternative to using GSD, as in the case of building of web services on demand (Magdalenić, Radošević, & Skočir, 2009), which cannot be done easily by using the generic approach.

Taking into account the complementariness of the different afore-mentioned paradigms, a number of authors (e.g. Fuentes, Nebrera, & Sánchez, 2009; Groher & Voelter, 2009) have integrated two or more approaches within a single approach that would yield significant synergy effects. With the objective of contributing to the SPLE body of knowledge we initiated research towards the autogeneration of software. Our approach is primarily based on Generative programming and Frame Technology with some crucial adjustments, like the usage of dynamic frames generation (Radošević & Magdalenić, 2011b; Radošević et al., 2012).

## 3. SCT model of source code generator

In the SCT generator model a generator is defined by three kinds of elements: Specification (S), Configuration (C) and Templates (T) (Radošević & Magdalenić, 2011b). Specification contains features of the generated application in form of attribute–value pairs. Templates contain the source code in a target programming language together with connections (in form of replacing marks) for insertion of variable code parts. Connections in Templates are replaced with other code templates or values from Specification in the process of source code generation. Configuration defines the connection rules between Specification and Templates, i.e. it determines how the source code generator is related with connections in the templates. All the three model elements together constitute the SCT frame (Fig. 1):

The process of source code generation starts with the initial SCT frame that contains the complete Specification and Configuration, and only the base template from the set of all Templates. Other

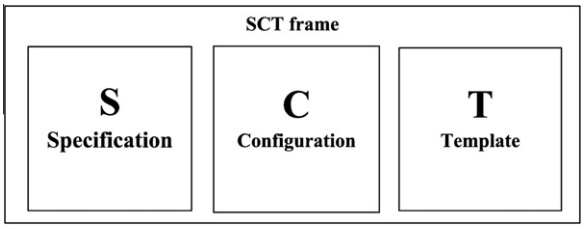


**Fig. 1.** SCT frame.

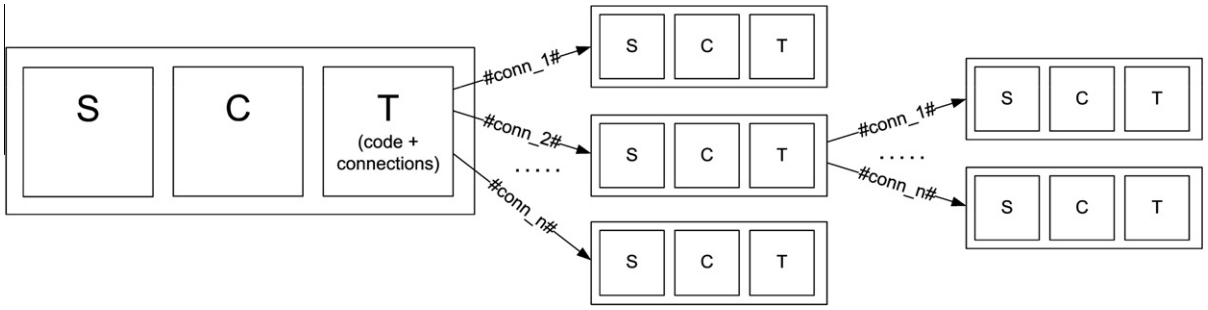


**Fig. 2.** Generation tree.

SCT frames are produced dynamically for each connection in the template, forming a generation tree (Fig. 2):

Although Specification and Configuration of new frames are inherited from their parent frames, there is also a possibility of filtering the elements of Specification and expanding the elements in Configuration (Radošević & Magdalenić, 2011b). The depth of the generation tree depends on Configuration rules. The process of source code generation is demonstrated on a simple example presented in Fig. 3. Specification defines the name of the generated program file *prog.py* and contains definitions of two variables *a* and *b*, both of which come with a description (i.e. subordinated attribute *desc*). Configuration defines the starting template *main.template* and three connections. Connections *variables* and *StringToInt* are used for the insertion of program templates *input.template* and *stringtoint.template*, respectively, for each occurrence of the definition of *variable* in Specification. The connection *variable name* is used for inserting a value of an element with the name *variable* from Specification. The template *main.template* has two connections named *variables* and *StringToInt*. The template *input.template* contains the connection *variable name* which occurs twice in the template. The template *stringtoint.template* contains connections *variable name* and *desc* that occur once for each occasion of the attribute *variable*. The generated code is presented as *prog.py*. The process of source code generation starts with *main.template*, where the connections named *variables* are replaced twice (once for each variable) with *input.template*. The connection *StringToInt* is also replaced twice with *stringtoint.template*.

Although in the presented simple example only one program file is produced, there is a possibility of generating more files from the same Specification. For this purpose, the SCT generator model defines a Handler (Radošević & Magdalenić, 2011b), i.e. a program that manages the generation of different program pieces from the same set of Specification, Configuration and Templates.

## 4. The model of Autogenerator

The model of Autogenerator (Radošević et al., 2012) is developed as an extension of the existing SCT model of the source code generator (Radošević & Magdalenić, 2011b), as presented in Fig. 4. This section offers a comprehensive description of all the steps in the autogeneration process.

A user request is accepted by a Request handler, whose purpose is to decompose the request and to determine what action to take. The Request handler has to build the initial SCT frame and call the source code generator to assemble the appropriate source code. It is notable that the aim of the SCT source code generator is changed when it is used as part of Autogenerator. The original SCT source code generator is designed to generate the source code of the entire application. When the SCT source code generator is used in Autogenerator, it produces only the source code that is needed to fulfil the user request, which differentiates it from the typical use of Generative programming where the source code of a complete application is generated. This is achieved by filtering of Specification, i.e. by taking only a subset of Specification. It is common for Specification to contain information needed for generating the source code of a complete application. By taking only a subset of Specification it is possible to generate the source code needed for certain actions. After the initial SCT frame has been built with the Specification subset, the source code generator is invoked. 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. 4 is intended for executing the generated source code together with arguments issued by the Request handler. Those arguments are presented in Fig. 4 as the Application context and are usually obtained from the user request. For example, the argument can refer to some information about the user who is performing a certain action or the 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 the user request. In the case of a web application, the response comes in form of a webpage to be presented in a web browser. In some cases the Execution unit performs the Specification update. This is an important feature of Autogenerator since it enables the application definition to be changed in runtime.

The creation and use of Autogenerator are described by means of roles performed by particular actors. The Use Case diagram of Autogenerator is shown in Fig. 5. There are three roles identified in the autogeneration process: domain expert, generator developer and application user. The role of the domain expert is to analyse the domain in which Autogenerator will be used. The domain expert has to identify user requirements. Based on the collected user requirements it needs to provide a list of features that the final application may contain. It should be noted here that we are dealing with a family of applications, which means that the final application is likely to have only one subset of all possible features depending on a particular user request. The next step performed by the domain expert is the construction of one or more application prototypes. In building of prototypes different ways of feature implementation are expressed. The created prototypes are analysed together with the generator expert for the purpose of source code templates development/design. Source code templates are pieces of source code to be embedded into the generated application. In construction of source code templates no general rules apply. For example, a source code template could function as the body of a method or, alternatively, as a piece of code, i.e. the declaration of a variable. It is important to identify source code templates that can be used for building different features of the final application. This enables the source code templates to be reused, which is the desired outcome of the process. Given that code templates represent basic inputs in the next step of the autogeneration process, collaboration between the domain expert and the generator developer is more than welcome.

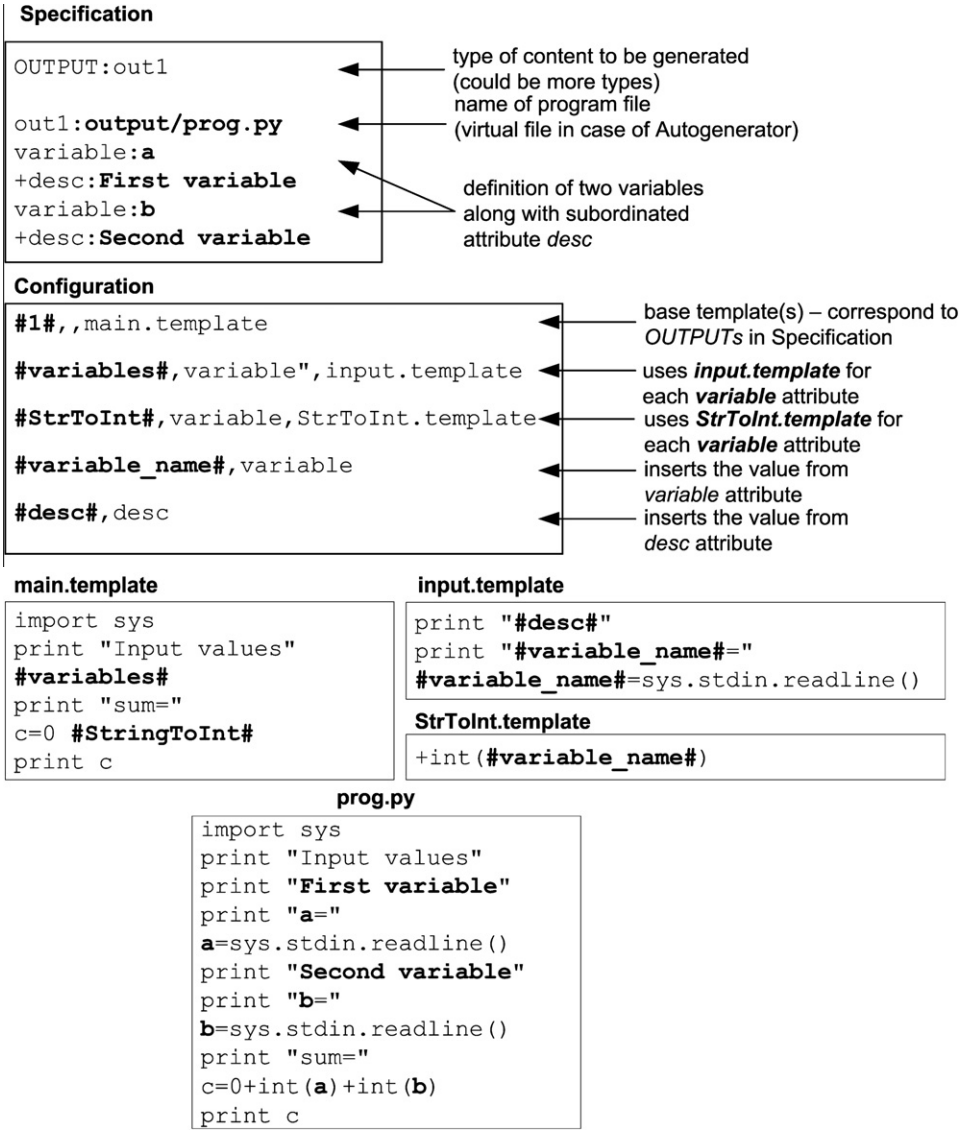


**Fig. 3.** Example of source code generation.

The next step in the process is building of the source code generator. The inputs to this process are the previously created source code templates on the one hand and the collected knowledge about user requests and desired application features on the other. The generator developer creates Specification and Configuration of the source code generator and inserts links in source code templates. This process is rather demanding and susceptible to errors. Consequently, to minimize errors and to help generator developers we have created two tools that can significantly speed up this process: error messaging tool and introspection tool.

The error messaging tool is described in detail in Radošević, Magdalenić, and Orehovački (2011) and serves to detect typical errors that occur in this phase of the source code generator creation. Typical errors include the following: the attribute in the application specification is declared but not used; the source code template is defined in the configuration but it does not exist; the source code template contains links that are not defined in the configuration. The introspection tool provides the information about which part of Specification and Configuration is used for fulfilling a particular service requested by the user as well as that concerning which source code templates are being used for that purpose. This tool is especially important in the building of Autogenerator where it can be used as kind of a debugging tool. An example of introspection tool use is shown in Section 5.5.

With regards to the generator architecture, a Request handler represents the specific difference between Autogenerator and other SCT based generators. As presented in Fig. 4, a Request handler is a part of Autogenerator which receives and decomposes the user request in order to determine what action to take and what the input parameters of that action are. As already stated, the Request handler has to build the initial SCT frame and call the source code generator to produce the appropriate source code. Based on the data singled out from the request, the Request handler has to determine the application context to be passed to the Execution unit. The application context stores the information about the user session in course of the action that requires several request/response iterations. If Autogenerator is used for implementation of a web application, the application context could contain information about the user's web session, connections to the database and the required environmental variables.

Finally, the role of the user encompasses sending the request and receiving responses. Consequently, the application has to be organized so that it operates on the request/response principle. Since most web applications actually work in that way,

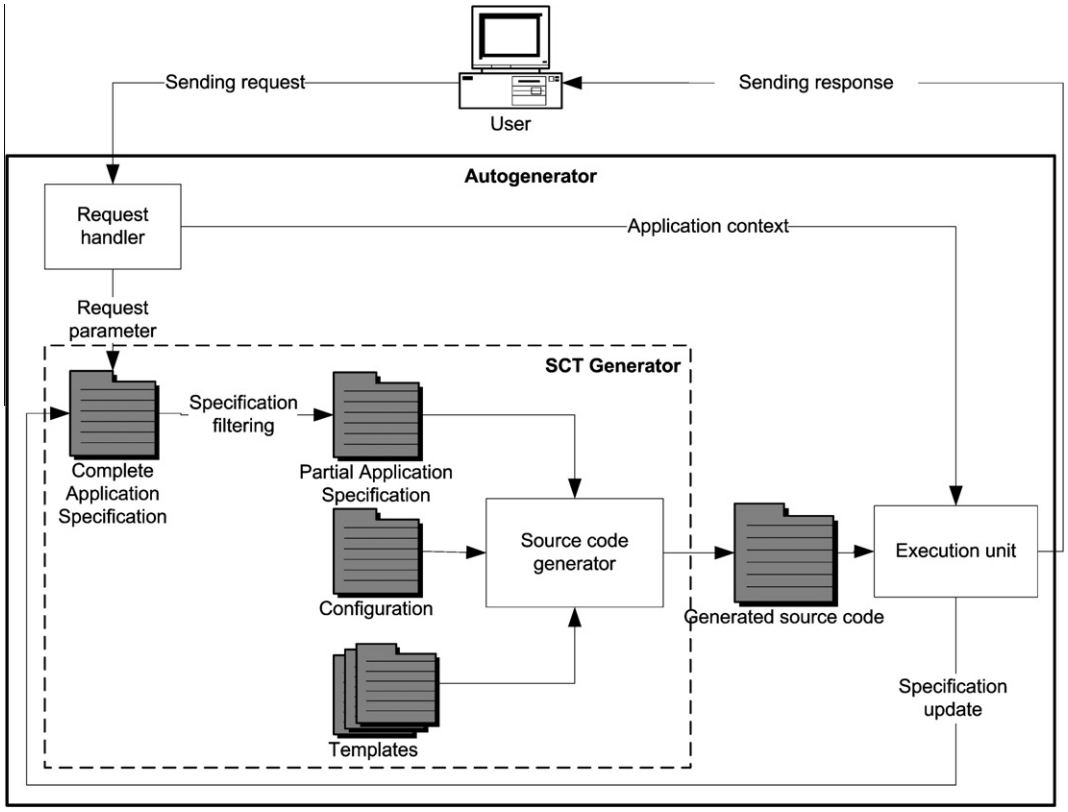


**Fig. 4.** Model of Autogenerator.

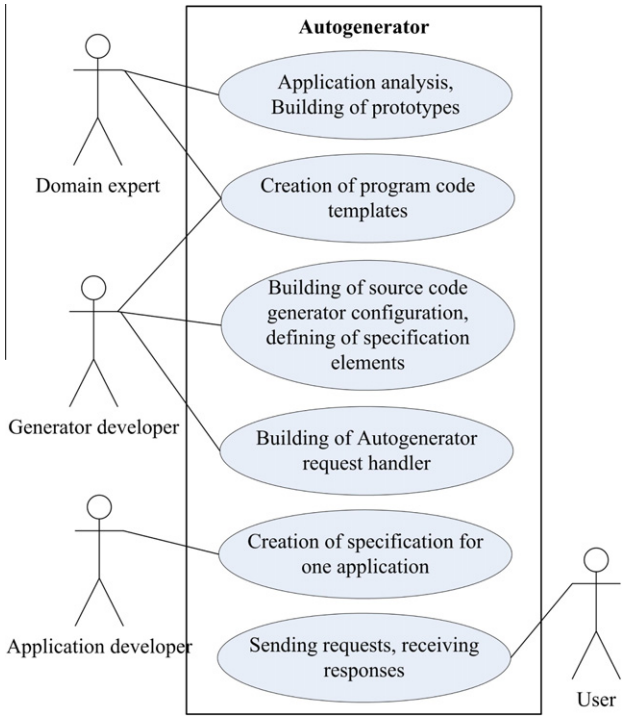


**Fig. 5.** Autogenerator Use Case diagram.

Autogenerator is particularly suitable for their implementation as it also enables a level of optimization to be obtained. Obviously, there is no need to generate the whole application for each user demand. With the aim of maintaining compatibility, the Specification of any SCT based generator includes names of program files to be generated which are contained in Autogenerator. In case of Autogenerator, these files are entirely virtual, so filenames are used only to name the generated pieces of the program code. Autogenerator uses the parameter named *file* in case of a web application which could be sent by using the GET method. The parameter *file* determines the piece of code to be generated and therefore also the part of Specification to be used. However, one particular piece of code is sometimes insufficient to perform a user action, e.g. showing a data entry form requires two pieces of code: program script and entry form. To address that issue, the generated piece of code has to be inspected to find filenames used in Specification, which means that such code pieces should also be generated.

A typical lifecycle of the Autogenerator development is presented in Fig. 6. The development starts with the building of one or more prototypes. Since the functionality of the final application is entirely contained in the application prototype, it should be noted that the application based on Autogenerator is intended for implementation of families of similar applications. After each application prototype has been analysed, general and specific characteristics of the application are defined. The application prototype is then decomposed into its components, some or all of which can be contained in the final application. The next step is the development of the source code generator. Each application component is separated as one template of the program source code. The configuration of the SCT generator determines how these templates should be put together. The specification of the SCT generator defines which components are to be included in a concrete application. Until this stage the development of an application based on Autogenerator does not differ from that of an ordinary application based on Generative programming principles. It is next step that is typical of Autogenerator-based applications. The program code templates need to be adopted in such a way that they support the Autogenerator model. This means that the application needs

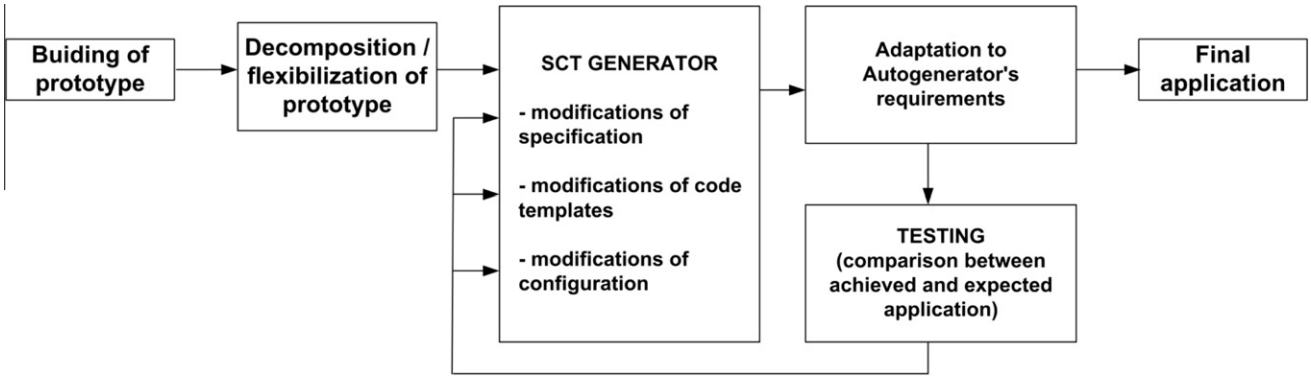


**Fig. 6.** Lifecycle of Autogenerator-based application development.

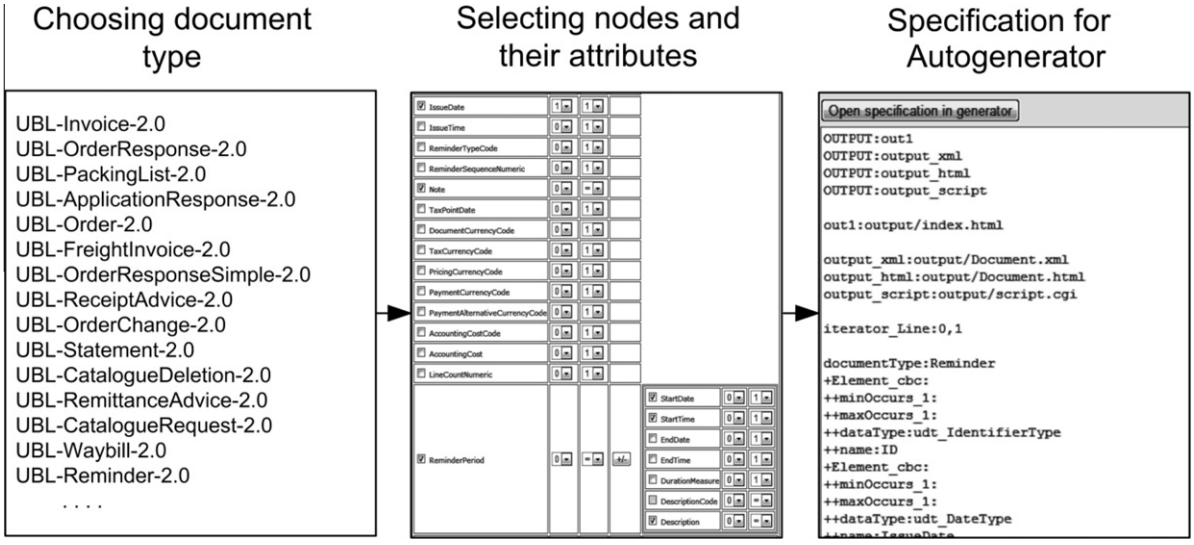


**Fig. 7.** Process of creating Specification.

to be organized to operate on the request/response principle and that additional information has to be included in requests and responses for the purpose of defining the action to be performed by Autogenerator. As mentioned above, in case of a web application such additional information can be a parameter file which determines the piece of code to be generated, and therefore, part of Specification to be used. The following step in application development is testing, where the achieved application is compared with its expected counterpart. If the application does not act as expected, it is necessary to go back to the SCT generator development stage. Modifications of specification, configuration and code templates are performed until the desired functionality is achieved. A similar approach to the development of applications that starts from prototypes is found in Heradioa, Fernandez-Amorosb, de la Torrec, and Abada (2012), where prototypes are referred to as exemplars.

The implementation of Autogenerator is discussed in more detail in the following section.

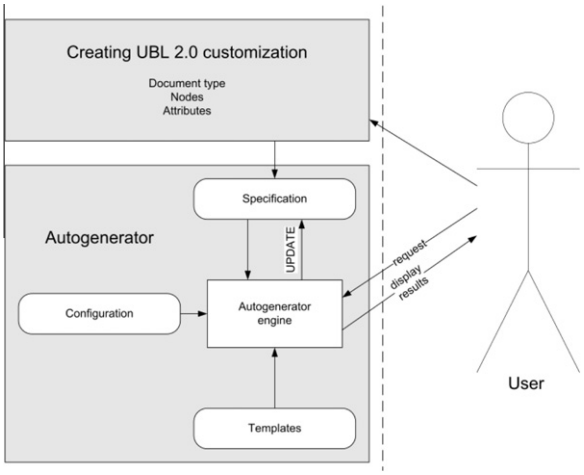


**Fig. 8.** Creating and using of UBL 2.0 customization.

## 5. Example of Autogenerator

As an example of Autogenerator[1,2] a web application is given that handles documents under the UBL 2.0 specification (OASIS, 2006). In the first part of the application, the user creates an UBL customization by defining XML nodes and their attributes that are used in particular UBL documents. This phase results in a Specification that Autogenerator uses in dealing with UBL documents (editing, displaying, storing into database, etc.; see Fig. 7). In our paper, Autogenerator uses a given Specification to produce the program code for manipulating UBL 2.0 documents, although a possibility of modifying Specification 'on the fly', as described further, also exists.

[1] <http://gpml.foi.hr/ubl/>.
[2] <http://gpml.foi.hr/SCT_Autogenerator_Example/>.

### 5.1. Creating UBL 2.0 customization

The first phase, which encompasses the creating of UBL 2.0 customization, results in Specification that Autogenerator uses in the dynamic (i.e. one performed individually on each user request) generation of a web application. The user chooses the type of a UBL document (such as invoice, order, reminder, etc.) as well as appropriate nodes and their attributes that result in the Specification (Fig. 7), which is used by Autogenerator (Fig. 8).

### 5.2. Basic SCT model elements

Autogenerator uses the same SCT model elements (Specification, Configuration and Templates) as a conventional generator that produces program files (Fig. 9). All three model elements define the top-level frame, while all the other frames are instantiated dynamically in the generation process.

#### 5.2.1. Creating Specification

Specification consists of attributes and their values. The first few lines of Specification specify the types of content to be generated (output types). These lines are connected to initial lines in Configuration that specify the usage of top-level code templates. Each output type could be used in the generation of specific content, e.g.:

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

demands generation of *index.html* (in case of Autogenerator this name only refers to the content, not a real file).

Attributes that contain *iterator_* in their name (e.g. *iterator_Line*) are used to specify ascending or descending values. All the other attributes could be used to specify different values, or/and as containers, e.g.:

```
documentType:Reminder
+Element_cbc:
++minOccurs_1:
```

specifies that the value of *document_type* is *Reminder*, *Element_cbc* is subordinated to *documentType* and *minOccurs_1* is subordinated to *Element_cbc* (values can be omitted).

An example of Reminder Specification is provided below:

```
OUTPUT:out1
OUTPUT:output_xml
OUTPUT:output_html                  Output types (index page, XML document,
OUTPUT:output_script                HTML form and CGI script)


out1:output/index.html
output_xml:output/Document.xml      Virtual output files (to retain compatibility
output_html:output/Document.html    with regular SCT generator)
output_script:output/script.cgi

                                    Range of values starting from 0 with step 1
iterator_Line:0,1                   (used for index values of array containing XML document)

documentType:Reminder
+Element_cbc:
++minOccurs_1:
++maxOccurs_1:
++dataType:udt_IdentifierType
++name:ID
+Element_cbc:
++minOccurs_1:
++maxOccurs_1:                      Specification of Reminder:
++dataType:udt_DateType             - Element_cbc = particular document item
++name:IssueDate                    - minOccurs_ = minimal number of occurences
+Element_cac:                       - maxOccurs_ = maximal number of occurences
++minOccurs_0:                      - dataType = UDT data type
++maxOccurs_unbounded:              - name = name of the element
++name:ReminderPeriod               - Element_cac = container for Element_cbc
+++Element_cbc:
++++minOccurs_0:
++++maxOccurs_1:
++++dataType:udt_DateType
++++name:StartDate
+++Element_cbc:
++++minOccurs_0:
++++maxOccurs_unbounded:
++++dataType:udt_TextType
++++name:Description
```

The example specifies four output files and four virtual files to be generated. Iterator is used in the generation of an array containing an XML document. The type of document under UBL 2.0 to be generated is *Reminder*. The document has a hierarchical structure where *Element_cac* acts as the container for other elements (*Element_cac* and *Element_cbc*).

*5.2.2. Creating Configuration*

Configuration of the example consists of simple rules that lead the Autogenerator engine in assembling of program code to be executed. The first few lines of Configuration are connected to output types in Specification. These lines define top-level code templates to be used in code generation, i.e. all types of content to be generated from Specification (e.g. HTML code, cgi scripts, Javascript code, etc.). All the other rows in Configuration consist of three elements separated by a comma: link (replacing mark in '#'-es) that occurs in code templates, attribute from Specification and the code template. The last element can be omitted, e.g.:

```
#name#,name
```

which requires that all occurrences of *#name#* are replaced by the value of the *name* attribute in Specification. Otherwise, specifying the third element requires that the code template is used, e.g.:

```
#cac_content#,name,cac_content.template
```

which in turn requires that all occurrences of *#cac_content#* are replaced by the content of *content.template* as many times as the *name* attribute occurs in Specification.

The asterisk is used to specify groups of attributes or templates, e.g.:

```
#cac_and_cbc#,Element_*,Element_*.template
```

where *Element_** is a group of attributes (e.g. *Element_cac* and *Element_cbc*) and *Element_*.template* is a group of files (e.g. *Element_cac.template* and *Element_cbc.template*).

The complete Configuration of the example is as follows:

```
#1#,,index.template
#2#,,invoice.template
#3#,,html_form.template
#4#,,ScriptContent.template
```
**Top-level templates** (connected to output types in Specification)

```
#imperatives#,ADD_field_*,ADD_field_*.template
#ADD_field#,ADD_field_*
#imperatives#,DELETE_field_*,DELETE_field.template
#DELETE_field#,DELETE_field_*
```
**Imperative instructions** (capital letters are being removed from Specification after execution)

```
#content#,documentType,content.template
#cac_and_cbc#,Element_*,Element_*.template
#name#,name
#cac_content#,name,cac_content.template
#documentType#,documentType
```
Document type, element type (*cac* or *cbc*, element name, usage of templates

```
#dataType#,dataType
#minoccurs#,minOccurs_*,minOccurs_*.template
#maxoccurs#,maxOccurs_*,maxOccurs_*.template
```
Data type and number of occurrences

```
#html_content#,documentType,HtmlContent.template
#html_cac_and_cbc#,Element_*,HtmlElement_*.template
#html_cac_content#,name,HtmlContent.template
```
Usage of HTML templates

```
#xml_content#,documentType,xml_content.template
#xml_cac_and_cbc#,Element_*,xml_Element_*.template
#xml_cac_content#,name,xml_cac_content.template
```
Usage of XML templates

```
#script_content#,documentType,ScriptContent.template
#script_cac_and_cbc#,documentType,script.template
#table_header#,field_*,table_header.template
#field#,field_*
#display#,display
#rows_to_variables#,field_*,rows_to_variables.template
#print#,field_*,field_print_*.template
#field_print#,field_*
#fields#,list(field_*)
#empty_fields#,field_*,empty_*.template
```
Used in generation of CGI script in Python

comma-separated list


**Fig. 9.** Basic model elements in *html* form.

The example above specifies four top-level templates that are connected to output types in Specification. Imperatives are used for changing the database table structure, as described in Section 5.4.2. Some links like *#name#* and *#documentType#* are common for different types of the generated content. Other specific groups of links also exist that are aimed for the generation of HTML (and contained Javascript), XML and CGI script in Python.

### 5.3. Assembling of program code

Assembling of program code depends on user parameters given by the *GET* method:

- *file*: the name of the virtual file to be generated
- *action*: the name of the user-requested action (e.g. display, data entry, correction, delete, etc.)

The generated code is stored into a dictionary (a Python data structure), where keys are names of virtual files, while values contain the generated code for execution. The user parameter *file* is used to determine which piece of code should be generated. There could be some dependent files, e.g. an *html* form connected with a *cgi* script. Autogenerator searches the generated code in order to identify each occurrence of a virtual file that is given in Specification. These virtual files are generated into a dictionary. To work properly, all file-related operations (such as opening, reading and writing the file, etc.) have to be redirected to appropriate methods that work with the dictionary.

It is important that in the process only the part that depends on user parameters (i.e., *file* and *action* in our example) needs to be regenerated, while regenerating the entire application on each user request is not necessary.

### 5.4. Achieved features

There are three basic features of Autogenerator that are not present in common generators: changing the application 'on the fly', imperative statements and code introspection. These features are implemented in the example provided in this paper.

#### 5.4.1. Changing the application 'on the fly'

Changing the application 'on the fly' is the first and most obvious feature of Autogenerator that arises from the avoidance of program files. Any change in Specification is updated instantly during execution without the need of restarting the application. The entire program code needed for the execution of user request is regenerated into variables (for which a Python dictionary is used) on each user request. Since no program files could possibly remain after some previous execution there is no need to clean them either. Configuration and Templates could also be modified 'on the fly', enabling even a substantial change in the application structure. However, this approach does not comply with the basic principle of the SCT generator model, which proposes the extraction of variable program features into Specification.

#### 5.4.2. Imperative statements

Imperative statements in Specification are used to perform certain instructions only once, which are 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 of program code. For example, the Specification statements:

```
ADD_field_text:new_field      – adding new field
+display:New field            – subordinated field
```

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_text:new_field       – established new state
+display:New field
```

Imperative statements have to be included in Configuration:

```
#imperatives#,ADD_field_*,     – usage of template
  ADD_field_*.template
#ADD_field#,ADD_field_*        – field name
#imperatives#,DELETE_field_*,  – usage of template
  DELETE_field.template
#DELETE_field#,DELETE_field_*  – field name
```

The link *#imperatives#* is used in the appropriate code template (here: in the main *cgi* script template). If there are no imperatives (*ADD* or *DELETE*) in Specification, the link *#imperatives#* will be replaced with blank. In case of usage of some *ADD field_** (e.g., *ADD_field_text*), the appropriate code template will be used, such as *ADD_field_text.template*:

```
sql="alter table UBL_table add     – SQL query
  #ADD_field# varchar"
try:
  cur.execute (sql)                – executing
                                   query
  conn.commit ()
  sql="update UBL_table set        – setting
  #ADD_field#=' ' where ID>=0"     initial value
  cur.execute (sql)
  conn.commit ()
  update_specification             – updating
  ("ADD","update")                 specification
except:
  print "Can't alter table
  ⟨b⟩UBL_table⟨/b⟩!⟨br⟩⟨br⟩\n"
  sys.exit (0)
```

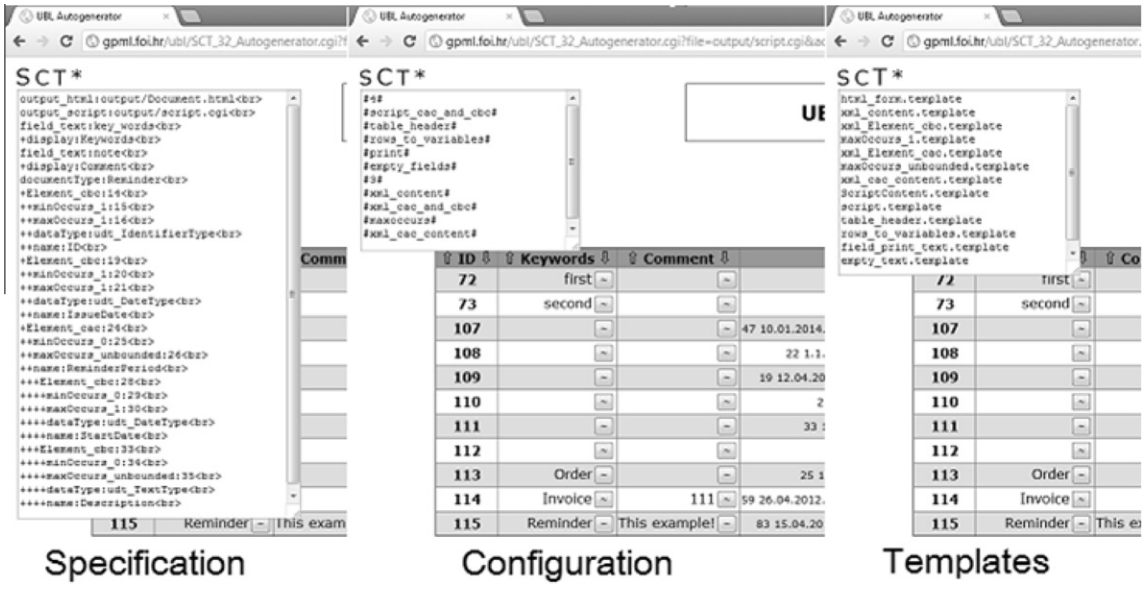


**Fig. 10.** Introspection pane.

This code uses the link *#ADD_field#*, which will be replaced by the field name. The function *update_specification* is embedded in Autogenerator.

### 5.5. Code introspection

Code introspection in Autogenerator enables application developers to see exactly which part of Specification, Configuration and Templates was used in the generation of a currently executed part of an application. In the example application, introspection is implemented in form of an introspection pane, as shown in Fig. 10.

The introspection is based on the fundamental Autogenerator principle that there is no need to regenerate the entire application on each user request. The generated code depends on user parameters (*action* and *file* in the example) that designate the needed parts of Specification, Configuration and Templates which can be presented to the user/developer.

## 6. Performance evaluation

The generation process, according to the SCT model, consists of reading files (Specification, Configuration and a subset of Templates), creation of generation trees (by dynamic allocation of SCT objects; see Section 3) and assembling of program code. The phase of saving the generated code into program files is void since Autogenerator executes the code from variables. As a result of the generation phase, the execution of programs in form of Autogenerator is generally slower than the execution of handwritten or previously generated programs. The aim of this section is therefore to investigate main factors that impact the duration of the generation process as well as of the response time of web applications executed in form of Autogenerator. The Autogenerator performance was compared with the performance of previously generated web applications that achieve the same functionality. It is important to note that the presented results are preliminary owing to a relatively small research sample and lack of some real-life situations, like multiple concurrent users, server overload, etc.

### 6.1. Experiment setup

We deployed both our Autogenerator prototype and the appropriate previously generated application of the same functionality on an edge server (Blade server, 4X Intel (R) Xeon (R) CPU E7-4850 @ 2.00 GHz, 12 GB RAM, 120 GB HDD) running a virtual machine under Debian GNU/Linux 6.0. The database server is run on a separate machine (same processor as the edge server, 3 GB RAM) under Debian GNU/Linux 6.0.5, running PostgreSQL 8.3.17. The edge server uses the Apache 2.0.49 Web server with Python 2.6.6. Both Autogenerator and the previously generated application were implemented in form of CGI scripts in Python.

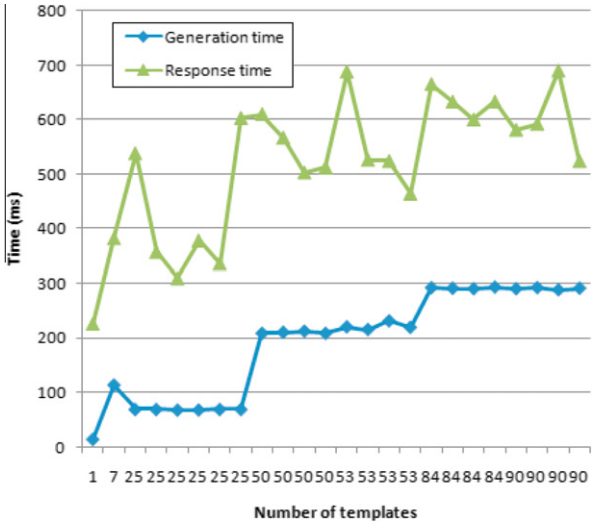


**Fig. 11.** Correlation between generation/response time and number of used code templates.

The duration of the generation process was measured using Python functions from *time* library, achieving the precision in 1/1000 s. The time interval that was measured started at the beginning of the Autogenerator execution and finished at the end of code generation. The execution time of the generated code was not measured at this point. The response time of both web applications in form of Autogenerator and the previously generated application was measured for different user actions (e.g., display database table content, form data entry) using the appropriate plug-in for the Chrome browser (page speed test[3]). Broadband internet connection was used (3072/192 kbs) in the process.

[3] Available at: <https://chrome.google.com/webstore/>.

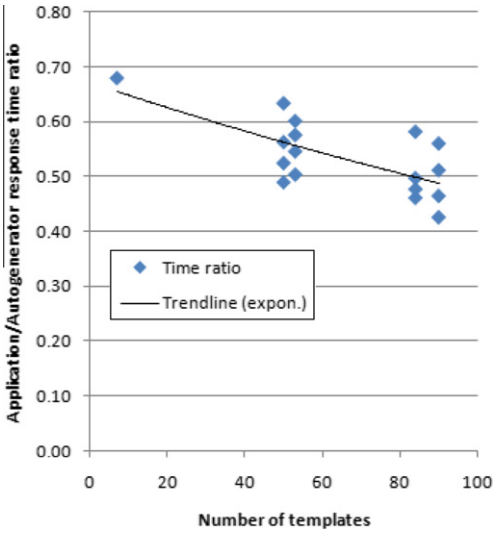


**Fig. 12.** Autogenerator slowdown expressed by response time ratio.

### 6.2. Generation and response time

The Autogenerator generation and response time were measured alongside the number of used templates. Generation time is the time needed to produce program code from SCT model elements. Response time is the overall time that consists of generation time, execution time and broadcast time. Generation time is measured on a server, while response time is measured on a client (Web browser). As shown in Fig. 11, a strong and significant Pearson's correlation ($r = .95$, $p < .01$) was found between the generation time and the number of used code templates. The Pearson's correlation established between the response time and the number of used code templates was also strong and significant ($r = .76$, $p < .01$).

This conditionality mostly arises from the demanding memory allocation operation because the new SCT object has to be allocated for each used code template. The number of used unique code templates was also measured, yielding its high and significant correlation with the generation time ($r = .93$, $p < .01$). On the other hand, results suggest that the size of the used code templates has no significant influence on generation time ($r = -.35$ ns). The templates used in test were in the range from 1 to 35 kB.

### 6.3. Response time of Autogenerator and previously generated application

When compared to a previously generated application of the same functionality, it is evident that using Autogenerator slows down program execution. This difference arises from the generation phase that is included in each operation to be performed. In our experiment, the response time of Autogenerator was measured alongside the response of the previously generated application of the same functionality, with the same operations performed in both applications (e.g. showing index page, table review, data entry, etc.). The comparative performance test also resulted in a response time ratio (application/Autogenerator response time), which is always lower than 1. The particular ratio depends on the complexity of generation, expressed in the number of used code templates. As shown in Fig. 12, the response time ratio is close to 1 in case of less complex generations and lower than .5 for more complex generations (e.g. those implying the usage of more than 50 code templates).

Generally speaking, the expected response time of Autogenerator could be approximately twice as long as that of a previously generated Web application of the same functionality, although the complexity of generation needs to be taken into account.

## 7. Conclusion

This paper presents the model of Autogenerator aimed at generating program code and executing it on demand. The presented model heavily relies on the previously introduced SCT model of the source code generator. SCT is based on dynamic frames that are formed and allocated during the generation process, unlike other frames-based generator models.

There are several benefits for application developers and users provided by Autogenerator. The most obvious one is the possibility to change the application 'on the fly' by changing its Specification or even Configuration and code templates. This feature can be exploited in the development process and debugging. Imperative statements can be used to perform some rarely used instructions (like changing of a database table structure) together with Specification update, after a new state of the application has been achieved. Finally, introspection gives developers feedback about specific pieces of Specification, Configuration and Templates that were used to perform a requested operation. However, in addition to the benefits of using Autogenerator, there are some limitations and even disadvantages of such an approach. Firstly, the concept is closely connected to scripting languages that have the possibility of evaluating code from variables. Secondly, the autogenerated application should also be in a scripting language, preferably the same one as the generator.

The example of the application given in our paper is applicable to documents under the UBL 2.0 specification. It encompasses the creation of UBL 2.0 specifications, creating/editing of documents, editing of the database table structure using imperative statements, as well as introspection and the possibility of changing the application 'on the fly'. The presented application is a representative example of the Autogenerator scope. Autogenerator is especially suitable for areas in which a lot of variants are required and where usual solutions based on parameterization could be very complex. The Autogenerator performance depends primarily on the complexity of generation. Since the generation of program code consists of reading files, creating generation trees and assembling of program code, optimizations are possible in all of these components. Our preliminary performance evaluation has shown that there is a strong and significant correlation ($r = .95$; $p < .01$) between the generation time and the number of used code templates. In comparison to the previously generated web application, the response time yielded by Autogenerator was approximately twice as long, although this also depends on the number of used templates.

It can therefore be concluded that we have shown that the idea of program code generation and its execution on demand is possible and represents a promising line of research. Our future work will focus on the development of a new software architecture based on a repository of reusable model elements. We believe that dynamic code generation has the potential to provide additional flexibility for software development and maintenance.

## References

Bassett, P. G. (1997). *Framing software reuse – lessons from real world*. Prentice Hall.
Bassett, P. G. (2007). The case for frame-based software engineering. *IEEE Software, 24*(4), 90–99.
Batory, D., Singhal, V., Thomas, J., Dasari, S., Geraci, B., & Sirkin, M. (1994). The GenVoca model of software-system generators. *IEEE Software, 11*(5), 89–94.
Classen, A., Heymans, P., & Schobbens, P.-Y. (2008). What's in a feature: a requirements engineering perspective. *Lecture notes in computer science* (Vol. 4961, pp. 16–30). .
Clements, P., & Northrop, L. (2002). *Software product lines: Practices and patterns*. Boston: Addison-Wesley.

Czarnecki, K. (2005). Overview of generative software development. *Lecture notes in computer science* (Vol. 3566, pp. 326–341). .

Czarnecki, K., & Eisenecker, U. W. (2000). *Generative programming methods, tools, and applications*. Boston: Addison-Wesley.

Czarnecki, K., Kim, C. H. P., & Kalleberg, K. T. (2006). Feature models are views on ontologies. In *Proceedings of the 10th international software product lines conference* (pp. 41–51). Baltimore: IEEE Press.

De Lara, J., & Vangheluwe, H. (2004). Defining visual notations and their manipulation through meta-modeling and graph transformation. *Journal of Visual Languages and Computing, 15*, 309–330.

Emrich, M., & Schlee, M. (2003). CodegenerierungmitXFramer und Programmiertechnikenfür Frames. *Objektspektrum, 5*, 55–60.

Fowler, M. (2010). *Domain-specific languages*. Boston: Addison-Wesley.

Fuentes, L., Nebrera, C., & Sánchez, P. (2009). Feature-oriented model-driven software product lines: the TENTE approach. In E. Yu, J. Eder, C. Rolland (Eds.), *Proceedings of the forum of the 21st international conference on advanced information systems (CAiSE), Amsterdam* (pp. 67–72).

Greenfield, J., & Short, K. (2004). *Software factories: Assembling applications with patterns, models, frameworks, and tools*. Indiana: Wiley Publishing.

Grigorenko, P., Saabas, A., & Tyugu, E. (2005). Visual tool for generative programming. In *Proceedings of the joint 10th European software engineering conference (ESEC) and the 13th ACM SIGSOFT symposium on the foundations of software engineering (FSE-13)* (pp. 249–252). ACM Publication.

Groher, I., & Voelter, M. (2009). Aspect-oriented model-driven software product line engineering. *Lecture notes in computer science* (Vol. 5560, pp. 111–152). .

Grossman, I., & Mah, M. (1994). Independent research study of software reuse using frame technology. Technical Report, QSM Associates.

Guo, S., Tang, L., & Xu, W. (2010). XVCL – an annotative approach to feature-oriented programming. In *Proceedings of the 2010 international conference on computational intelligence and software engineering* (pp. 1–5). Wuhan: IEEE Press.

Haase, A., Völter, M., Efftinge, S., & Kolb, B. (2007). Introduction to openArchitectureWare 4.1.2. <http://www.voelter.de/data/workshops/oawSubmissionToolsWorkshop.pdf>.

Heradioa, R., Fernandez-Amorosb, D., de la Torrec, L., & Abada, I. (2012). Exemplar driven development of software product lines. *Expert Systems with Applications, 39*(17), 12885–12896.

Jarzabek, S., Bassett, P., Zhang, H., & Zhang, W. (2003). XVCL: XML-based variant configuration language. In *Proceedings of the international conference on software engineering* (pp. 810–811). Los Alamitos, CA, USA: IEEE Computer Society.

Jarzabek, S., & Zhang, H. (2001). XML-based method and tool for handling variant requirements in domain models. In *Proceedings of the fifth IEEE international symposium on requirements engineering* (pp. 166–173). Toronto: IEEE Press.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., & Peterson, A. S. (1990). *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report, Software Engineering Institute, Carnegie Mellon University.

Kang, K. C., Lee, J., & Donohoe, P. (2002). Feature-oriented product line engineering. *IEEE Software, 19*(4), 58–65.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold, W. (2001). Getting started with AspectJ. *Communications of the ACM, 44*(10), 59–65.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., et al. (1997). Aspect-oriented programming. *Lecture notes in computer science* (Vol. 1241, pp. 220–242). .

Lai, A., Murphy, G. C., & Walker, R. J. (2000). 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>.

Lemaire, C. (2010). *CodeWorker parsing tool and code generator – user's guide & reference manual, release 4.5.4*. <http://www.codeworker.org/CodeWorker.pdf>.

Loughran, N., Rashid, A., Zhang, W., & Jarzabek, S. (2004). Supporting product line evolution with framed aspects. In *Workshop on aspects, components and patterns for infrastructure software*. <http://www.comp.lancs.ac.uk/computing/aop/papers/SPL_ACP4IS2004.pdf>.

Magdalenić, I., Radošević, D., & Skočir, Z. (2009). Dynamic generation of web services for data retrieval using ontology. *Informatika, 20*(3), 397–416<http://www.mii.lt/informatica/htm/INFO755.htm>.

Műller, J., & Eisenecker, U. W. (2008). 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*. <http://www.infosun.fim.uni-passau.de/spl/apel/McGPLE2008/papers/Paper8.pdf>.

Minsky, M. (1975). A framework for representing knowledge. In P. Winston (Ed.), *The psychology of computer vision*. New York: McGraw-Hill.

Neto, V. V. G., & de Oliveira, J. L. (2011). An early aspect for model-driven transformers engineering. In *Proceedings of the 2011 international workshop on early aspects* (pp. 7–11). Porto de Galinhas: ACM.

OASIS, (2006). Universal business language 2.0. <http://docs.oasis-open.org/ubl/cs-UBL-2.0/UBL-2.0.html>.

Olson, A. M., Raje, R. R., Bryant, B. R., Burt, C. C., & Auguston, M. (2005). UniFrame: a unified framework for developing service-oriented, component-based, distributed software systems. In Z. Stojanovic, & A. Dahanayake (Eds.), *Service-oriented software system engineering: Challenges and practices* (pp. 68–87). Idea Group Publishing, Hershey, PA [Chapter IV].

Radošević, D., & Magdalenić, I. (2011a). Python implementation of source code generator based on dynamic frames. In N. Bogunović, & S. Ribarić (Eds.), *Proceedings of the 34th MIPRO international convention* (pp. 369–374). Opatija: MIPRO.

Radošević, D., & Magdalenić, I. (2011b). Source code generator based on dynamic frames. *Journal of Information and Organizational Sciences, 35*(2), 73–91.

Radošević, D., Orehovački, T., & Konecki, M. (2007). PHP scripts generator for remote database administration based on C++ generative objects. In *Proceedings of the 30th MIPRO international convention on intelligent systems* (pp. 167–171). Opatija: MIPRO.

Radošević, D., Orehovački, T., & Stapić, Z. (2010). Automatic on-line generation of student's exercises in teaching programming. In *Proceedings of the 21st Central European conference on information and intelligent systems* (pp. 87–93). Varaždin: FOI.

Radošević, D., Magdalenić, I., & Orehovački, T. (2011). Error messaging in generative programming. In *Proceedings of the 22nd Central European conference on information and intelligent systems (CECIIS) 2011, Varaždin, Croatia, 21st–23rd September 2011* (pp. 181–187).

Radošević, D., Orehovački, T., & Magdalenić, I., 2012. Towards software autogeneration. In *Proceedings of the 35th MIPRO jubilee international convention on intelligent systems, Rijeka, Croatia* (pp. 1257–1262).

Radošević, D., Konecki, M., & Orehovački, T. (2008). Java applications development based on component and metacomponent approach. *Journal of Information and Organizational Sciences, 32*(2), 137–147.

Rosenmüller, M., Siegmund, N., Saake, G., & Apel, S. (2008). Code generation to support static and dynamic composition of software product lines. In *GPCE'08: Proceedings of the 7th international conference on Generative programming and component engineering*.

Schlee, M., & Vanderdonckt, J. (2004). Generative programming of graphical user interfaces. In *Proceedings of the working conference on advanced visual interfaces* (pp. 403–406). Gallipoli: ACM.

Stahl, T., & Völter, M. (2006). *Model-driven software development: Technology, engineering, management*. West Sussex: Wiley & Sons.

Štuikys, V., & Damaševičius, R. (2000). Scripting language open PROMOL and its processor. *Informatica, 11*(1), 71–86.

Tolvanen, J. P., & Rossi, M. (2003). Metaedit+: defining and using domain-specific modeling languages and code generators. *OOPSLA 2003 demonstration*.

Warmer, J. B., & Kleppe, A. G. (1998). *The object constraint language: Precise modeling with UML*. Boston: Addison-Wesley.

Yuan, L., Song Dong, J., & Sun, J. (2006). Modeling and customization of fault tolerant architecture using object-Z/XVCL. In *Proceedings of the 13th Asia Pacific software engineering conference* (pp. 209–216). Kanpur: IEEE Press.

Zhang, H., & Jarzabek, S. (2004). XVCL: a mechanism for handling variants in software product lines. *Science of Computer Programming, 53*(3), 381–407.