UDC 004.4'242
Original Scientific Paper

Source Code Generator Based on Dynamic Frames

Danijel Radošević *danijel.radosevic@foi.hr*
University of Zagreb
Faculty of Organization and Informatics Varaždin

Ivan Magdalenić *ivan.magdalenic@foi.hr*
University of Zagreb
Faculty of Organization and Informatics Varaždin

Abstract

This paper presents the model of source code generator based on dynamic frames. The model is named as the SCT model because if its three basic components: Specification (S), which describes the application characteristics, Configuration (C), which describes the rules for building applications, and Templates (T), which refer to application building blocks. The process of code generation dynamically creates XML frames containing all building elements (S, C ant T) until final code is produced. This approach is compared to existing XVCL frames based model for source code generating. The SCT model is described by both XML syntax and the appropriate graphical elements. The SCT model is aimed to build complete applications, not just skeletons. The main advantages of the presented model are its textual and graphic description, a fully configurable generator, and the reduced overhead of the generated source code. The presented SCT model is shown on development of web application example in order to demonstrate its features and justify our design choices.
Keywords: dynamic frames, generative programming, specification, configuration, template

1. Introduction

Recent advances in Software Engineering have reduced the cost of coding programs at the expense of increasing the complexity of program synthesis, i.e. the process of coming up with the final program. Software Product Lines (SPL) and Model Driven Development (MDD) are two cases in point [23]. SPL provides a means for composing software products that match the requirements of different application scenarios from a single code base and can be developed using a variety of implementation techniques [19]. The well-known concepts in this area are Generative Programming [3], pre-processor definitions, components, Aspect Oriented Programming, Feature-Oriented Programming (FOP) [16], [19], Aspectual Feature C Modules (AFMs) [1] and frames like XVCL [25]. Using SPL helps to increase the software making productivity, by producing it in a way comparable to industrial production.This paper describes our approach to the synthesis of a program from a set of artifacts using a model specially developed for this purpose. In our case artifacts are code templates and application parameters which are synthesized according to the configuration of the source code generator.
Even today there is a lack of appropriate graphic and aspect based models intended for making and documenting of source code generators. The Specification-Configuration-Templates (SCT) model of a source code generator represents the current state of our intention to devise a source code generator model which is aspect based (as opposed to generic models), uses code Templates in the production of the source code, enables both graphic and textual representation of the generator, and is independent from the target programming language and problem domain. Furthermore, the SCT model has some other important features. The source code generator is defined as a multi level tree structure where a higher-level generator is given by the superposition of lower-level generators. This allows for the nesting of generators, similar to the nesting of program structures in structured and

Object-Oriented Programming. More generators can share the same application specification, giving different generated outputs (depending on the code templates used). Moreover, some polymorphic features are enabled similar to late binding of virtual methods in Object-Oriented Programming. A configuration can propose late binding of code Templates during the process of generation, which makes it easier to add new features to program Specification. Finally, modifications of generators are envisioned on all three model elements (Specification, Configuration and Templates) enabling simultaneous development of generators and generated applications.

The SCT generator model is primarily designed for web application development. Technically, there are no constraints to using the SCT model in development of any kind of a source-code generator, but web applications have some characteristics which make SCT-based generators suitable for their generation. Web applications usually consist of a larger number of small program units, called scripts (cgi scripts, php scripts, Java classes, etc.) that are suitable for generation from the same program Specification. The SCT based generators consist of a number of small generators that share the same Specification and generate different types of outputs. Furthermore, in most cases web applications already represent some kinds of generators (e.g. those of the HTML, XML or JavaScript code) that could lower the number of generation levels in the generator itself.

1.1. History of the model

The model is based on a previously introduced Scripting Generator Model (SGM) [17]. The primary aim of the SGM model was to make a suitable graphic generator model for modelling of generators written in scripting languages like Perl and PHP, since UML is intended for modelling applications in Object-Oriented Programming languages like C++ and Java. In addition, it was shown that SGM could also be used in the modelling of generators written in Object-Oriented Programming languages, e.g. by using C++ Generative Objects [18]. The generator model in SGM is much simpler than the corresponding UML model and the implementation of aspects, i.e. features that are not closely connected to individual program organizational units, like functions or classes [10], is much easier [18]. The model is extended by adding some polymorphic features, similar to dynamic polymorphism in Object-Oriented Programming.

The SCT generator model inherits the graphic diagrams for representing program Specification and Configuration from SGM. However, SCT offers a formal definition of the model and new textual model representation based on XML. Furthermore, Configuration was separated from the generator itself, meaning that the complete generator problem domain can be changed without any changes in the generator code. This could upgrade the previous approaches to making generators and making applications using generators toward real Generative Programming [3], where these two processes are joined together.

1.2. Paper outline

Following the presentation of related work in Section 2, the definition of the SCT model and its XML and graphic representation are given in Section 3. Section 4 gives an example of a web application generator. The conclusion is given in Section 5.

2. Related work

There are several programming disciplines in automatization of programming which share similar goals and/or approaches as our research. Generative Programming (GP) is a discipline of Automatic Programming introduced in the late 1990's that deals with designing and implementing software modules [3]. Modules can be combined to generate specialized and highly optimized systems fulfilling specific requirements [5]. Basically, GP uses advanced concepts from Object-Oriented Programming (OOP), like Generic Programming, together with Metaprogramming, Domain Engineering and especially Aspect Oriented Programming

(AOP). AOP focuses on the modularization of crosscutting concerns in complex software [10]. Implementation of GP techniques should result in optimization, which most specifically differentiates it from other Automatic Programming techniques.

Jarzabek's XVCL is a frame mechanism based on Bassett's frames [8]. XVCL uses x-frames as building blocks of program code to be generated. These x-frames are organised in a tree structure, where specification x-frames (or SPC for short) contain program specification [2]. Other x-frames combine program code with break sections that define insertion of variable program parts (defined by other x-frames). Configuration elements are specified implicitly, in break sections, defining different kinds of insertion and adaptation. All used x-frames form a tree structure where SPC-s are on the top.

Generally, XVCL uses static frames that are all defined by developer. In SCT, frames are instantiated dynamically, during the process of generation. The SCT frames form a tree structure, where each frame contains clearly separated parts regarding to Specification, Configuration and code template (particular template from Templates). Templates contain typeless connections instead of break sections in CVCL. This approach enables SCT to be more flexible in generative application development, because building of generation tree and usage of particular code templates depends on Specification, enabling additional possibilities, including polymorphic features (as described in section 3.3.2. Polymorphic Configuration elements).

Our approach is similar to Feature Oriented Programming (FOP) [16], [19]. FOP treats software features as fundamental units of abstraction and composition. We use an application Specification where the features of the final application are defined, similar to FOP. Code templates, which are the main building blocks, contain connections which can be used for adding different crosscutting concerns. Finally, we use the Configuration of the source code generator to make problem-domain adjustments, which is done by pre-processor definitions in the approach given by Rosenmüller [19].

The SCT model is oriented to working with code-fragment-sized components. The same approach is used in [7]. Other GP based projects, like Uniframe [14], [24], avoid descending to code-fragment-sized components. Our components are not necessarily strictly connected to program organizational units, like classes or methods. Consequently, our approach differs from the metaclass-based approaches, as described by Grigorenko et al. [6], Tolvanen and Rossi [22] and De Lara and Vangheluwe [4].

Some approaches are based on manipulation or generation of programs within the language, which requires a language with metalanguage capabilities, i.e., the minimum ability being that of representing programs in the language itself. Languages like Lisp [20], MetaML [21], `C [15] and DynJava [13], provide such facilities. C++ provides a solution with template metaprogramming [3], where generated programs are expressed as parameterized types, and code is produced by a compiler through inlining [19]. Our solution is not based on the generation of programs within a language. Avoiding inlining enables the generation of program code in any programming language, depending on code Templates used. The used code Templates contain only one sort of replacing marks which are typeless. These replacing marks are replaced by the program code during the process of generation.

Although the SCT model can be used in the generation of a wide array of applications, some areas, such as web applications, are more suitable for it. We use our approach mainly in building web applications and web services [12]. Web applications are particularly suitable for SPL because of a high rate of source code repetition, which is also recognized in [9]. Empirical studies have shown 50-90% rates of repetitions that deliberately recurred in newly developed well-designed programs [9].

3. SCT generator model

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

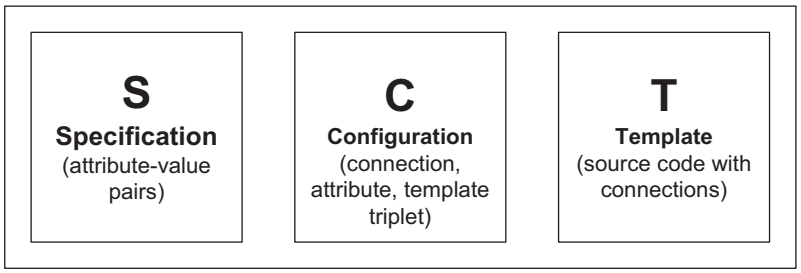

Fig. 1: SCT frame

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

Starting SCT frame contains the whole Specification, the whole Configuration, but only the base template from the set of all Templates. Other SCT frames are produced dynamically, for each connection in template, forming generation tree (Fig. 2):

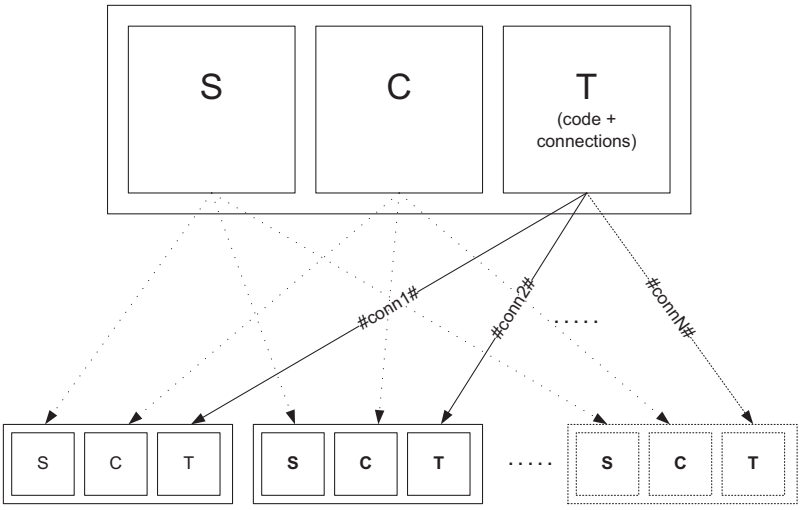

Fig. 2: The generation tree

Specification and Configuration of new frames are inherited from their parent frames in a general case, but there are possibilities of filtering (Specification) and expanding (Configuration; see 3.3.2. Polymorphic Configuration elements). The depth of generation tree depends on Configuration rules.

3.1. XML frames implementation

The XML frames implementation defines three types of elements:

- **Specification** (Specification of the generated application)
  - attribute-value pairs to specify features of the generated program
- **Configuration** (connection rules between Specification and Templates)
  - connections together with attributes and Templates
- **Templates** (code Templates in the target programming language)
  - target programming language code with connections (to be replaced by ***SCT*** structures)

The XML format is defined by XML Schema[1].

[1] http://generators.foi.hr/xml_schema.jpg

The appropriate XML document is as follows (Fig. 3):

```
<sct>
  <specification>
    <s attribute="..." value="..." role="..."/>
    <s attribute="..." value="..." role="..."/>
  </specification>
  <configuration>
    <c connection="..." source="..." template="..."/>
    <c connection="..." source="..." template="..."/>
  </configuration>
  <template>
  . . .
  </template>
</sct>
```

application specification

generator configuration

code template

Fig. 3: XML record of SCT frame

This is the top-level generator representation. Templates include connections to lower levels, each of them representing the whole SCT frame (Fig. 4):

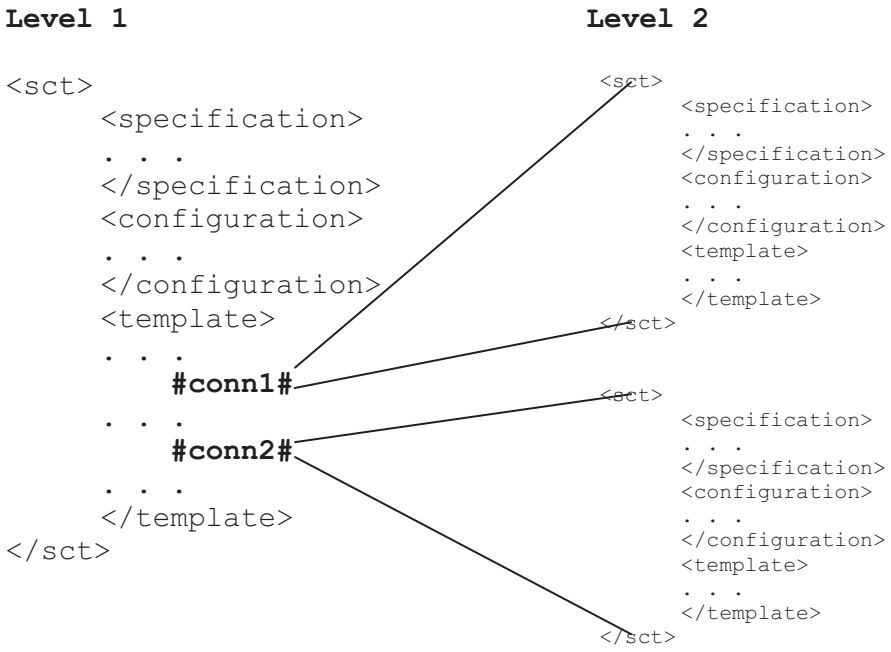

Fig.4: Connections to lower-level SCT frames

The lower-level frames are created dynamically by the automated process of generation, where each lower-level Specification and Configuration is derived from their higher level. Each template is loaded from an outside source (e.g. a textual file), according to appropriate Configuration rule. The role of the generator is to derive the starting top level frame into lower-level frames.

3.2. Specification

Specification defines application properties which fulfil the user needs. The purpose of the generator is to embed these properties into the generated code by associating them to Templates. Elements of Specification are *attribute-value* pairs:

```
<s attribute="attribute_name" value="attribute_value"/>
```

3.2.1. Containers

Some attributes are containers for other attributes, which are subordinated, e.g. the attribute *title* is a container for the attribute *title_display* in the following example:

```
<s attribute="title" value="students" role="container">
  <s attribute="title_display" value="FOI students"/>
</s>
```

The contained attributes cannot be used independently from their containers, because the usage of container requires the usage of the next-level template in Configuration (where a subordinated attribute is available). That prevents mixing of subordinated attributes by insuring that each of them really belongs to its container.

3.2.2. Groups

Groups are abstract (have no values) and enable uniform processing of similar attributes. In the following example, group *field_* contains two members (*field_int* and *field_char)*:

```
<s attribute="field_" role="group">
  <s attribute="field_int" value="student_id"/>
  <s attribute="field_char" role="surname_name"/>
</s>
```

These fields could be referred to in Configuration as *field_** (all members of *field_* group) or separately as *field_int* or *field_char*. All group members must contain the same prefix (here *field_*).

3.2.3. Outputs

Specification should define where the generated code should finish. This is done by the pre-declared attribute OUTPUT, which is used to define output types. Types of outputs are linked to base templates (as specified in Configuration) and used in specifying names of generated files. Outputs are a special kind of containers, which can share the same Specification to produce different kinds of output files, as shown in the following example (Fig. 5):

```
<specification>
  <s attribute="OUTPUT" value="script"/>
  <s attribute="OUTPUT" value="form"/>
  <s attribute="script" value="example.cgi"/>
  <s attribute="form" value="form.html"/>
  <s attribute="title" value="students" role="container">
   <s attribute="title_display" value="FOI students"/>
  </s>
   <s attribute="table" value="foi_students"/>
   <s attribute="primary_key" value="student_id"/>
  <s attribute="field_" role="group">
   <s attribute="field_int" value="student_id"/>
   <s attribute="field_char" value="surname_name"/>
  </s>
  <s attribute="script" value="script.cgi"/>
  . . .
</specification>
```

output types

output files

attribute values are shared among different outputs

next file of type *script* to be generated

Fig. 5: Output types and output files

Output types *script* and *form*, shown in Fig. 5, are linked to appropriate base templates in Configuration which refer to cgi script and html form to be generated. All subordinated attributes refer to both output files (*example.cgi* and *form.html*).

3.2.4. Specification diagram

Graphically, the hierarchy among Specification attributes can also be represented by a Specification Diagram [17], as shown in Fig. 6. The Specification diagram is a hierarchic diagram that defines the proposed application properties in the form of a tree-like feature model (for a similar approach, see Limbourg and Kochs [11].

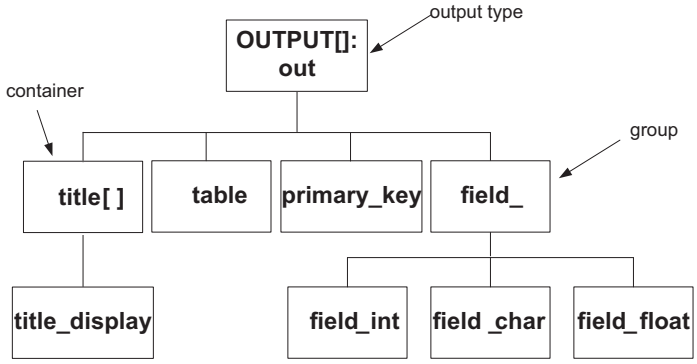

Fig. 6: Example of Specification Diagram

Attributes defined as containers are marked by '*[]*'. Groups have suffix '_'.

3.3. Configuration

Configuration defines connections between Specification and Templates. Configuration consists from Configuration rules. Each Configuration rule is defined by three elements:

- **Connection.** Each connection is physically placed inside Templates, and marked by '#' signs, e.g. *#title#*, defining the position where the real content should be placed. Connection is the key element that occurs once in Configuration, but one or more times in Templates.
- **Source.** Each connection has the appropriate source in Specification, e.g. the source for connection *#title#* is the value of attribute *title*. The source can be defined as: particular, container, or group.
- **Code template.** If the source is a container or a group, the connection has subordinated code template, otherwise, the element is omitted.

Connections used in code templates define inclusion of content that can be from another code template, or source, if code template is omitted. Recursive connections (leading to same template) have to be avoided. Similar to Specification, Configuration is organized hierarchically, which can be represented by XML notation or graphically, by a Configuration Diagram.

3.3.1. Specifying Configuration

Configuration is specified by three-element groups, containing:

- **connection** (base element),
- **source** (attribute from Specification) and
- **template** (lower level template, if present)

These groups are specified by the '***c***' element in the XML notation:

```
<c connection="..." source="..." template="..."/>
```

Connections occur in Templates, and should be replaced by the program code during the process of generation. The source refers to the value of a particular attribute or all values from a group to be used in code generation. The template can be omitted, which means that the connection should be replaced by the appropriate source. If the template is specified, then it should be used for each appearance of the specified source. These three elements are also used in specifying Configuration graphically (Fig. 7) by a Configuration Diagram (previously called the Metascript Diagram [17]).

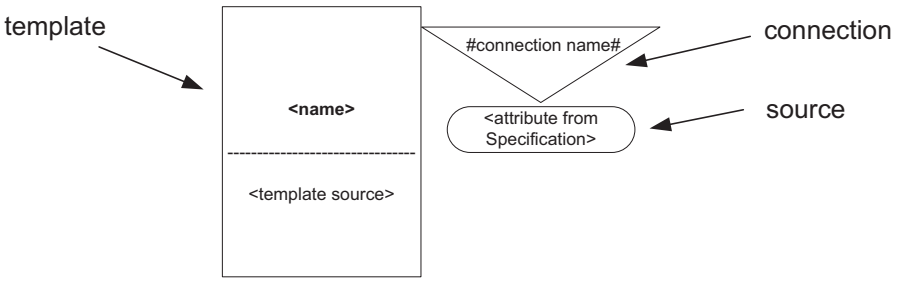

Fig 7: Elements of Configuration Diagram

Connection is the base element; sources and code templates are attached to connections, as shown in the following example (Fig. 8):

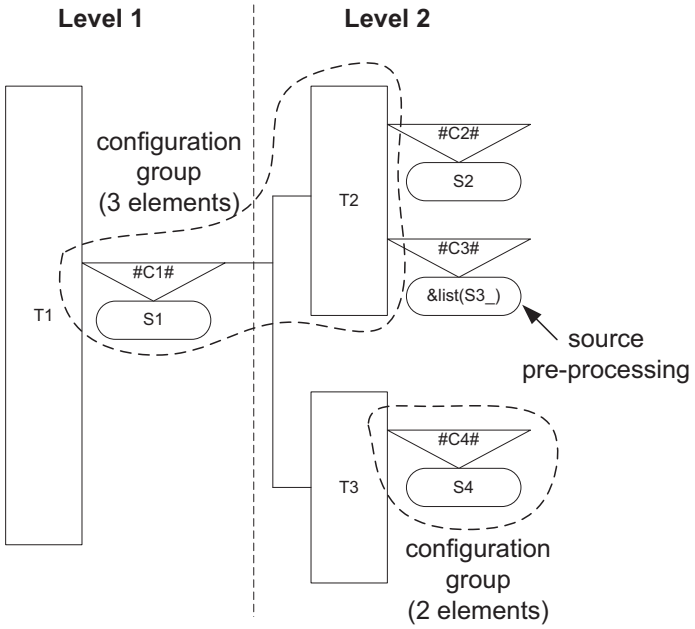

Fig. 8: Configuration groups

The base template (**T1**) is 'dangling' and has to be specified separately (Fig. 9).

Instead of a real connection, base templates have a number between '#' signs. The number represents the ordinary number of the connected output type in Specification. The group source ***S3_*** is pre-processed by the ***list*** process, which makes a list from values of attributes from ***S3*** group, as described further.

3.3.2. Source pre-processing

Source pre-processing converts Specification group to a particular source using some specialized function. In this way the exceptions in the model are manipulated when sources

from Specification group cannot be treated uniformly. A typical example is the field list in the SQL select query, e.g.:

```
SELECT student_id, surname_name, enrolment_year FROM
students
```

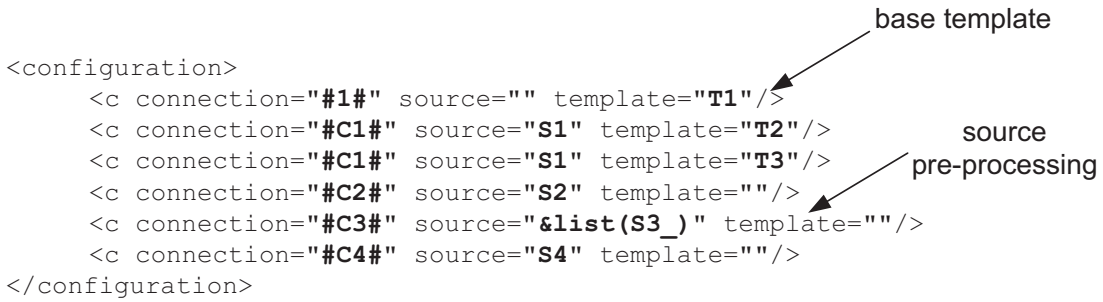

Fig. 9: Specifying of base template

While the whole select query could be represented by a single template, field names as a group source (*field_*) and table name as the value of the *table* attribute, there is still a problem with commas after the field names: note that there is no comma after the last field name.
The SCT model offers a solution in the form of source pre-processing, meaning that some sources should be pre-processed before their usage, which is marked by the '&' sign:

```
source="&<operation_name>(<group attribute>)"
```

Example:

```
<c connection="#list#" source="&list(field_)"
template=""/>
```

In the above example the list of fields is used as a particular source. As a result, each attribute value in the list ends with the ‘,’ sign, except for the last one, which is omitted.

3.3.3. Polymorphic Configuration elements

Configuration enables specifying of late binding of attribute values to Templates, which could rationalize the Configuration, e.g.:

```
<c connection="#field#" source="field_int"
template="field_int.template"/>
<c connection="#field#" source="field_float"
template="field_float.template"/>
<c connection="#field#" source="field_text"
template="field_text.template"/>
```

could be replaced by a single line:

```
<c connection="#field#" source="field_*" template="field_*.template"/>
```

meaning that each member of the *field_* group should be connected to the appropriate template with the same variable part of the name.

3.4. Templates

Physically, Templates are program code fragments which contain connections. Each SCT group contains only one template, but other SCT groups are included recursively via connections, as previously shown in Fig. 4. For example, Fig. 10 shows the XML representation of the code template which contains the basic structure of a web application index page (HTML):

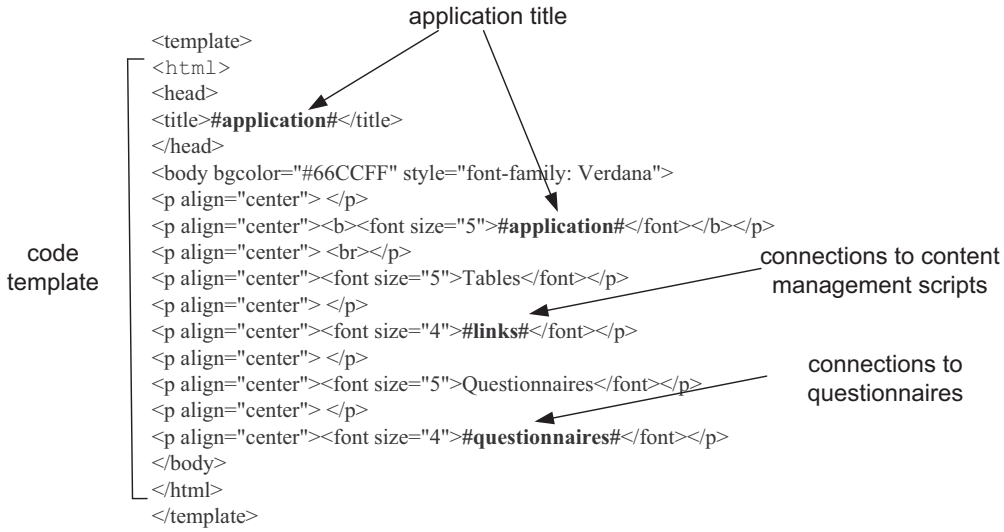

Fig. 10: Example of code template

As can be seen in Fig. 10, the template contains connections *#application#* (2 times), *#links#* and *#questionnaires#*. Each connection must be defined in Configuration (Fig. 11), together with the base template source (*index.template*):

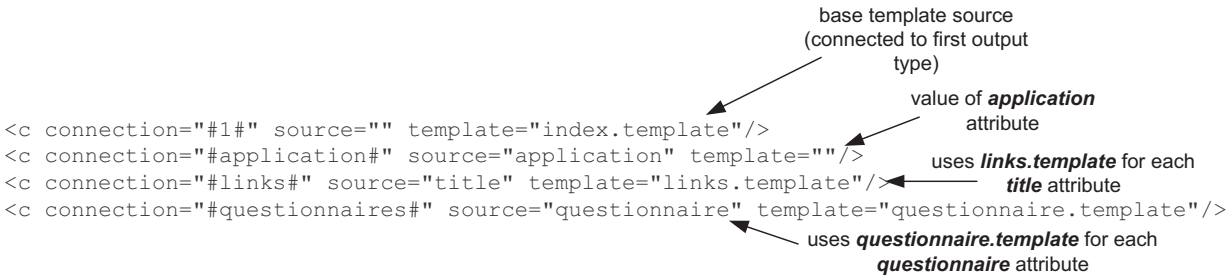

Fig. 11: Configuration lines connected to previous template

It is important that the same connection could be used in more than one template and more than once in a particular template, thus enabling a wide dispersion of Specification values without any change in Configuration.

3.5. Remaining issues

The SCT generator model defines the way in which a generator builds program code from three model elements: Specification, Configuration, and a set of Templates. However, there are still some 'what-if' issues concerning the implementation of the model, which should be resolved in future research. Some of those issues are related to, respectively, the consistency of the model implementation (e.g. that there are no attributes in Specification that are not included in Configuration), syntactic correctness of the generated code as well as its logical correctness. The main inconsistencies of model implementation to be checked are as follows:

- **Basic syntax.** Inconsistencies according to the XML scheme (invalid Specification or Configuration).

- **Specification.** Usage of attributes not included in Configuration or types of outputs not matching the initial part Configuration (defining base templates).
- **Configuration**
  - Referencing of non-existing templates. In some cases (e.g. in case of using polymorphic Configuration elements) it is possible to use replacing templates.
  - Usage of connections that do not appear in Templates, possibly due to a wrong connection or a redundant Configuration element.
- **Templates.** Usage of connections which are not included in Configuration.

The syntactic correctness of the generated code could, naturally, be checked by compiling. However, compiler error messages can sometimes confuse the developers. It is important to consider possible causes of syntactic incorrectness:

- **Insufficient Specification.** Some necessary attributes and their values are not specified, which causes that some connections stay unused in the generated code. This could be detected regardless of compiling: the necessary Specification attribute could be found in Configuration according to the unused connection in the generated code.
- **Usage of unsafe names in Templates** (variables, functions, classes etc.). A potential cause of syntactic incorrectness because the attribute values from Specification could collide with the names in Templates. Using names with prefixes/suffixes could reduce the risk.
- **Calls of functions prior to their declarations.** Some programming languages require that functions be defined prior to their calls. The order of Specification attributes could lead to the breach of that rule. This could be solved by providing function declarations prior to their use (which should be included in Templates or generated).

The logical incorrectness of the generated code could be caused by some of the following:

- **Usage of unsafe names in Templates** (variables, functions, classes etc.). Instead of causing syntactic incorrectness, unsafe variables could threaten the stability and correctness of the generated program during runtime.
- **Breaking program restrictions.** Exceeding the size limit and other restrictions caused by Specification values.

Generally, the issues can be avoided/solved by the appropriate generative application development process, where building generators and generated applications are closely connected processes. Tracking changes should help in finding/solving errors. Furthermore, some issues could be solved by introducing checks specific to the problem domain, e.g. restrictions in attribute values, number of attributes etc.

4. Example of a generator

The example generator[2] is made for the purpose of web application generation, which includes database content management and web questionnaires (Fig. 16). There are many generic content management systems which work on generic databases and use generic software instead of specific one. Our example generator generates cgi scripts (in Perl) that create database tables, performing major database operations (review of table content, adding/editing of records via html forms and deleting records), interconnection of database tables (lookup and master-detail connections) and creation of web questionnaires with a collection of results.

[2] http://arka.foi.hr/~darados/SCT_generator/

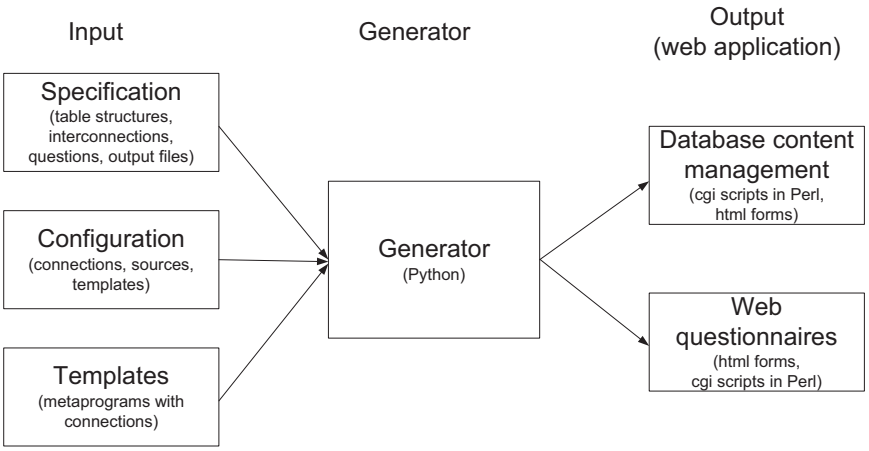

Fig. 16: Example generator

The example generator uses the textual form of Specification, Configuration and Templates.

4.1. Specification

Specification defines specific features of the generated application, which differentiates it from all the other applications from the same problem domain (as defined by the generator Configuration and Templates). The first part of the Specification defines the kinds of outputs to be generated, e.g.:

```
<s attribute="OUTPUT" value="index"/>
<s attribute="OUTPUT" value="output_cgi"/>
<s attribute="OUTPUT" value="output_html"/>
<s attribute="OUTPUT" value="questionnaire"/>
```

where *index* refers to the index page (html), *output_cgi* to Perl scripts for content management, *output_html* to html forms for content management and *questionnaire* to Perl scripts for maintaining questionnaires. The definition of the index page is the simplest part of Specification, e.g.:

```
<s attribute="index" value="output/index.html"/>
<s attribute="application" value="Database Content
Management"/>
```

where *index* is a kind of output, and *application* is an attribute (the value of *application* replaces the appropriate connection *#application#* in the template, as defined in Configuration). Note that not all the features of the index page are specified here, so the contained links are generated from the rest of Specification (Fig. 17).

The entity title with the corresponding table is defined by its name, table name, field names and other attributes, e.g. (Fig. 18):

Attributes defining table fields consist of the group name (*field_*) and suffix (e.g. integer or text) that defines the type. The questionnaire is defined as an extension to the table definition, by *questionnaire_name* and by attaching attributes to table fields (question, question type and offered answers) e.g. (Fig. 19).

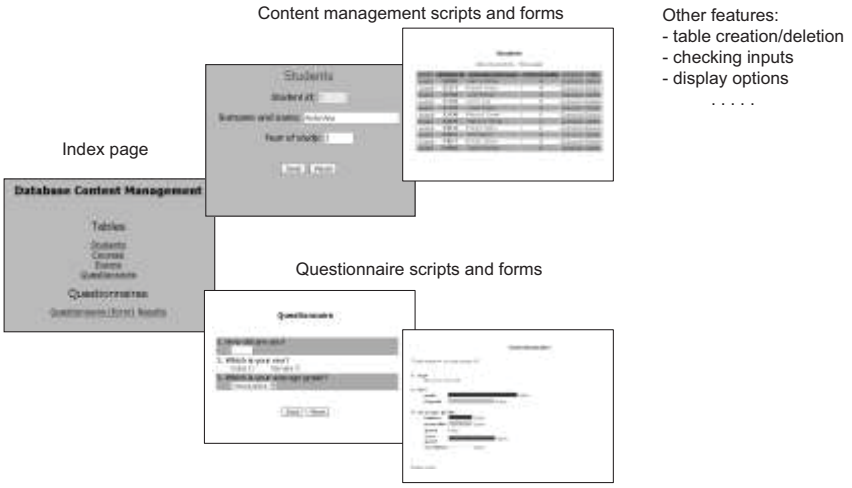

Fig.17: Structure of generated application

```
<s attribute="output_cgi" value="output/students.cgi"/>
<s attribute="output_html" value="output/students_form.html"/>
<s attribute="title" value="students" role="container">
  <s attribute="title_display" value="Students"/>
</s>
<s attribute="db" value="$DB=DBI->connect('dbi:Pg:dbname=db1;host=localhost','db1','');"/>
<s attribute="table" value="students"/>
<s attribute="connection" value="exams" role="container">
  <s attribute="conn_field" value="student_id"/>
</s>
<s attribute="primary_key" value="student_id"/>
<s attribute="field_" role="group">
  <s attribute="field_integer" value="student_id" role="container">
    <s attribute="field_display" value="Student id"/>
  </s>
  <s attribute="field_text" value="surname_name" role="container">
    <s attribute="field_display" value="Surname and name"/>
  </s>
  <s attribute="field_integer" value="year_of_study" role="container">
    <s attribute="field_display" value="Year of study"/>
  </s>
</s>
```

output files for different kinds of outputs

container

group

members of *field_*

Fig. 18: Specification of particular entity

```
...
<s attribute="questionnaire" value="output/questionnaire_form.cgi"/>
...
<s attribute="questionnaire_name" value="questionnaire1" role="container">
  <s attribute="questionnaire_name_display" value="Questionnaire (form)"/>
</s>
...
<s attribute="field_" role="group">
  <s attribute="field_integer" value="sex" role="container">
    <s attribute="field_display" value"Sex"/>
    <s attribute="question" value="Which is your sex?" role="container">
      <s attribute="question_radio" value="" role="container">
        <s attribute="answer" value="male"/>
        <s attribute="answer" value="female"/>
      </s>
    </s>
  </s>
</s>
...
```

output file

group (*field_*)

lowest attribute level

Fig. 19: Specification of Questionnaire

The attribute *question_radio* has no value, but it determines the template to be used in generation. Specification is graphically shown in the Specification Diagram (Fig. 20).

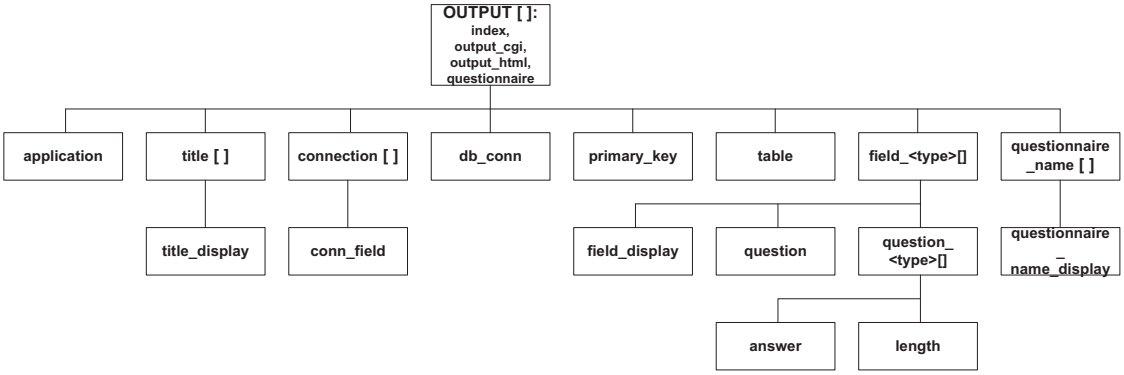

Fig. 20: Specification Diagram of example generator

As shown in the Diagram, there are four output types defined (*index*, *output_cgi*, *output_html* and *questionnaire*). Each output type can be used more than once in Specification, producing more output files (all of them sharing the same Specification).

4.2. Configuration

Configuration defines connections between the application Specification and Templates. In its first part, kinds of outputs are attached to their highest-level templates (Table 1):

Configuration	Specification
<c connection="#1#" template="index.template"/> <c connection="#2#" template="script.template"/> <c connection="#3#" template="form.template"/> <c connection="#4#" template="questionnaire.template"/>	<s attribute="OUTPUT" value="index"/> <s attribute="OUTPUT" value="output_cgi"/> <s attribute="OUTPUT" value="output_html"/> <s attribute="OUTPUT" value=" questionnaire"/>

Table 1: Kinds of outputs with their highest-level templates

The number between the '#' signs defines the ordinal number of the *output* kind. The rest of the Configuration defines three element groups where:

- the first element is a connection (physically present in Templates),
- the second element is an attribute name from Specification and
- the third element is the attached template (omitted if there is no need for a template)

For example, the line:

```
<c connection="#table#" value="table" template=""/>
```

means that the connection *#table#* should be replaced by the value of the attribute *table* from Specification in all their occurrences in the appropriate template. At the same time,

```
<c connection="#links#" value="title"
template="links.template"/>
```

means that connection *#links#* should be replaced by the whole template *links.template* for each occurrence of the attribute *title* (e.g. it is used for generating links on the index page). In case of group attributes from Specification, it could be specified as:

```
<c connection="#form_fields#" value="field_*"
template="field_form_*.template"/>
```

meaning that the connection `#form_fields#` should be replaced by the whole template for each occurrence of any attribute with a name starting with *field_* (e.g. *field_integer* or *field_text*). The template name is given by replacing the asterisk by field type (e.g. *field_form_integer.template*). In case of source pre-processing, it is specified as:

```
<c connection="#fields#" value="list(field_*)"
template=""/>
```

meaning that the connection *#fields#* should be replaced by the value created by function *list*. It uses all attributes with a name starting with a field to create the output value (e.g. it is usable for generating a field list in SQL queries). The order of Configuration lines is unimportant.

Configuration can be represented by a Configuration Diagram. Each Configuration Diagram belongs to the appropriate kind of output. For a questionnaire, it is as follows (Fig. 21):

In Fig. 21, a dashed line rectangle, *Question_type*, shows a polymorphic feature where the real template will be determined at the time of generation. Level 1 in Fig. 21 shows the main template specified in the initial part of *Configuration*. This level defines the main structure of the generated code. Level 2 works with database fields and attached questions. Level 3 defines the main structure of a particular question (i.e. arrangement of the text and controls). Level 4 refers to all the templates whose name starts with *question_*. The usage of a particular template depends on the used Specification attribute (e.g. attribute *question_radio* causes the usage of *Question_radio.template*). Level 5 defines the management of particular answers as values used in controls (like a radio button or a combo box).

4.3. Achieved features

The aim of the generator is to generate an application with all the required features from a relatively small specification. The example of a generated application includes: four output types (cgi scripts for content management and questionnaires, html forms); content management of four database tables; and one questionnaire.

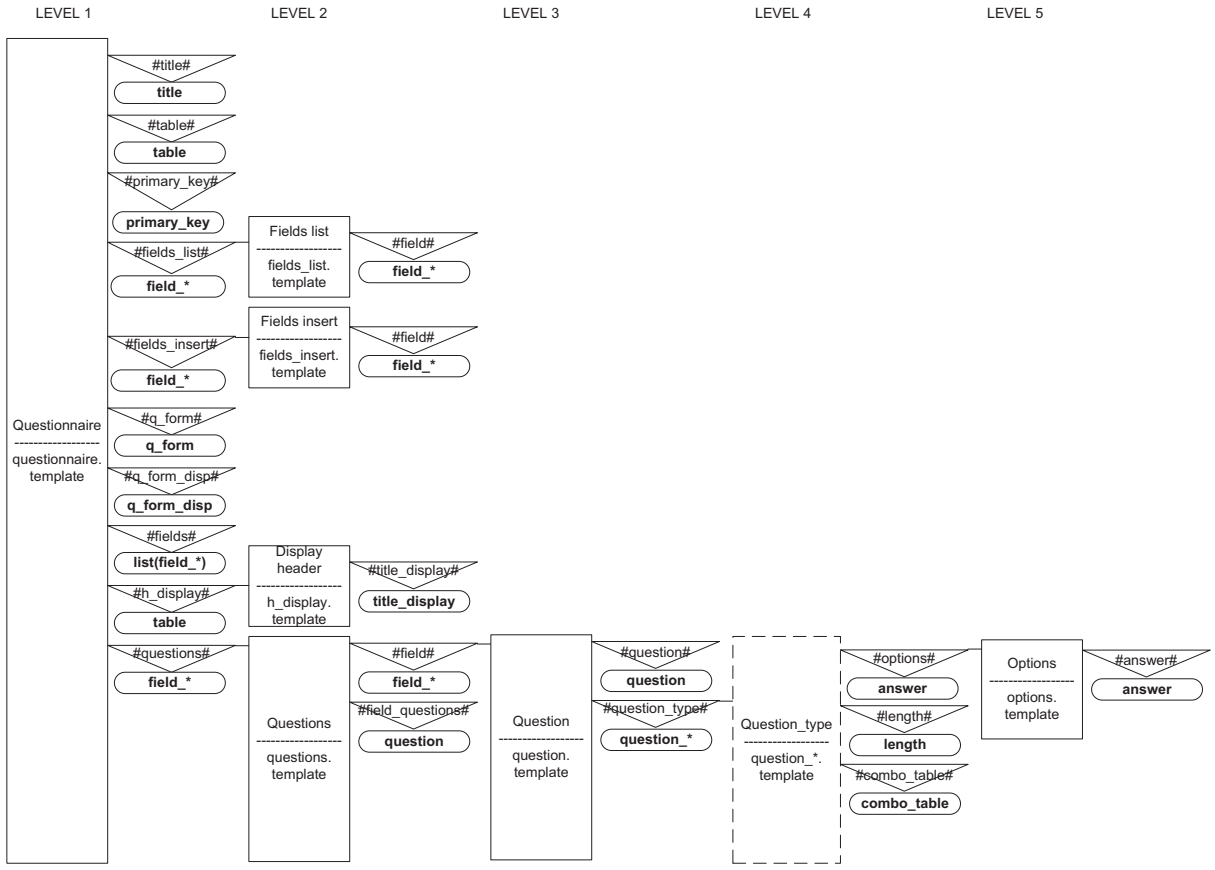

Fig. 21: Configuration Diagram for questionnaire

The generator inputs and outputs can be compared by the number of files and number of lines (Table 2):

Inputs			Outputs		
	lines	**files**		**lines**	**files**
Specification	114	1	**Generated code**	2627	12
Configuration	51	1			
Templates	790	57			

Table2: Generator inputs and outputs

The number of templates is bigger than the number of Configuration lines because a single Configuration line can define more than one physical template (e.g. *question_*.template* refers to more than one physical template).

4.3.1. Dispersion of connections and attribute values

Connections are included in Templates, defining variable parts of the program code to be generated. A single connection from Configuration may be contained in more than one template, more than once in a particular template, as shown in Table 3.

Configuration lines*	Occurrences in templates	Average connections per configuration line	Average connections per template
47	253	5,38	4,44

*Without lines that refer to output types

Table 3: Dispersion of connections through templates in example generator

Multiplying connections across Templates enables the dispersion of Specification values across the generated application, so a feature should only be defined once and can be used repeatedly. The usage of some Specification attribute values in the generated example application is shown in Table 4:

Attribute value	Attribute(s)	Total occurrences in generated code	Number of files where occur
age	**field_integer**	52	3
exam_date	**field_date**	30	2
exam_id	**field_integer, primary_key**	62	2
grade	**field_integer**	56	3
passable	**Answer**	2	1
questionnaire	**table, title**	56	4
student_id	**field_integer (students and exams), primary_key, combo_key**	114	4
students	**title, combo_table**	18	4
Students	**title_display**	4	3
t_courses	**Table**	17	1
year_of_study	**field_integer (students and courses)**	35	4

Table 4: Usage of some Specification attribute values in generated example application

4.3.2. Benefits for application updating

The multi-dispersion of connections could be used in application updating. Updating can be performed through changing of Specification, which enables new features of applications inside the problem domain proposed by Configuration. The updating of Templates changes the way Specification attribute values are used, including the programming language. The updating of Configuration changes the way the generator builds the program code. Introducing a new line in Configuration could enable the usage of a new Specification attribute and a new code template. This should make any later modifications of the generated code unnecessary.

5. Conclusion

This paper presents the SCT model of a source code generator for defining, building and documenting of a source code generator. The model consists of three components: Specification, which describes the application features, Configuration, which describes the rules for building applications, and a set of Templates, which are the main building blocks for generated applications. These three elements build SCT frames, while all SCT frames form a generation tree.

SCT was compared to XVCL. XVCL uses static frames that are all defined by developer. In SCT, frames are instantiated dynamically, during the process of generation, giving some additional possibilities, including polymorphic features.

The main advantages of the presented model are: the generator is fully configurable (the generator's source code does not have to be changed in order to change the generator problem domain); reduced overhead of the generated source code (only a subset of Templates is used depending on Specification); the generator is defined as a recursive multi-level tree structure; the solution is not tied to a programming language of the generated code; and both textual and

graphic model representation are present. SCT based generators are aimed to produce full applications or components and not merely skeletons to be finished afterwards.

The presented SCT model is shown on an application example to demonstrate its practical applicability and justify our design choices. Our future work shall focus on certain issues regarding the consistency check of the model implementation, e.g. its syntactic and logical correctness.

Acknowledgements

We are grateful to prof. Siniša Srbljić for his comments and insights on an earlier draft of this paper.

References

[1] Apel S., Leich T., Saake G. Aspectual Feature Modules. *IEEE Transactions on Software Engineering (TSE)*, 34(2):162–180, 2008.

[2] Blair J., and Batory D. A Comparison of Generative Approaches: XVCL and GenVoca. *Technical report, The University of Texas at Austin, Department of Computer Sciences*, December 2004.

[3] Czarnecki K,. Eisenecker, U.W. *Generative Programming: Methods, Techniques, and Applications.* Addison-Wesley, 2000.

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

[5] Eisenecker U. Generative Programming: Beyond Generic Programming, *Proc. Dagstuhl Seminar on Generic Programming,* April 27--May 1, 1998, Schloß Dagstuhl, Wadern, Germany, 1998.

[6] Grigorenko P., Saabas A., Tyugu E. Visual Tool for Generative Programming. *Proc. of the Joint 10th European Software Engineering Conference (ESEC) and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13).* ACM Publ., pp. 249–252, 2005.

[7] Griss M. L. Product line architectures. *In G. T. Heineman, & W. T. Councill (Eds.), Component-based software engineering: Putting the pieces together* (pp. 405-420). Boston: Addison-Wesley.

[8] Jarzabek S., Bassett P., Zhang H., and Zhang W. "XVCL: XML-based variant configuration language," in *Proc. Int'l Conf. on Software Engineering. Los Alamitos,* CA, USA: IEEE Computer Society, 2003, pp. 810–811.

[9] Jarzabek S., Pettersson U. Cost-Effective Engineering of Web Applications. *Proceedings of the 28th international conference on Software engineering*, Publisher: ACM, May 2006.

[10] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C. V., Loingtier J.-M., Irwin J. Aspect-Oriented Programming. *In Proceedings of the European Conference on Object-Oriented Programming (ECOOP),* volume 1241 of Lecture Notes in Computer Science, pp. 220–242. Springer Verlag, 1997.

[11] Limbourg P., Kochs H.D. Multi-objective optimization of generalized reliability design problems using feature models – A concept for early design stages, *Reliability Engineering & System Safety*, Volume 93, Issue 6, pp. 815-828, 2008.

[12] Magdalenić I., Radošević D., Skočir Z. Dynamic Generation of Web Services for Data Retrieval Using Ontology, *Informatika*, Volume 20 Issue 3, pp. 397-416, 2009. Available at: http://www.mii.lt/informatica/htm/INFO755.htm

[13] Oiwa Y., Masuhara H., Yonezawa A. DynJava: type safe dynamic code generation in Java. *JSST Workshop on Programming and Programming Languages*, PL2001, Tokyo, 2001.

[14] Olson, A. M., Raje, R. R., Bryant, B. R., Burt, C. C., Auguston, M. UniFrame: a unified framework for developing service-oriented, component-based, distributed software systems. *In Z. Stojanovic and A. Dahanayake (eds.), Service-Oriented Software System Engineering: Challenges and Practices* (Chapter IV, pp. 68-87). Hershey,PA: Idea Group Publishing.

[15] Poletto M., Hsieh W. C., Engler D. R., Frans Kaashoek M. `C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2), 1999, 324-369.

[16] Prehofer C. Feature-Oriented Programming: A Fresh Look at Objects. *In Proceedings of the European Conference on Object- Oriented Programming (ECOOP), volume 1241 of Lecture Notes in Computer Science*, pp. 419–443. Springer Verlag, 1997.

[17] Radošević D., Kliček B., Dobša J. Generative Development Using Scripting Model of Application Generator, *DAAAM International Scientific Book 2006*, DAAAM International, Vienna, Austria 2006.

[18] Radošević D., Orehovački T., Konecki M. PHP Scripts Generator for Remote Database Administration based on C++ Generative Objects, *Proceedings of the Mipro 2007*, Opatija 2007.

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

[20] Steele G. L. *Common Lisp the Language*, 2nd edition, Digital Press, 1990.

[21] Taha W., Sheard T. Multi-stage programming with explicit ennotations. *In Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97*, Amsterdam, pp. 203-217. ACM, 1997.

[22] Tolvanen J.P., Rossi M. Metaedit+: Defining and using domain-specific modeling languages and code generators. *In OOPSLA 2003 demonstration*, 2003.

[23] Trujillo S., Azanza M., Diaz O. Generative metaprogramming. *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, October 2007.

[24] Uniframe web site. http://www.cs.iupui.edu/uniFrame/, downloaded: September 1st 2009.

[25] Zhang H., Jarzabek S. XVCL: a mechanism for handling variants in software product lines, *Science of Computer Programming*, Volume 53, Issue 3 (December 2004) Pages: 381-407