Python Implementation of Source Code Generator Based on Dynamic Frames

D. Radošević, I. Magdalenić
Universitiy of Zagreb, Faculty of organization and informatics, Varaždin, Croatia
danijel.radosevic@foi.hr; ivan.magdalenic@foi.hr

Abstract - This paper presents the implementation in Python of the source code generator that is based on the SCT dynamic frames model. The SCT model consists of 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. Python is chosen as implementation language because of its flexibility as a scripting language with object-oriented features. The main advantages of the presented implementation are fully configurable generator, reduced overhead of the generated source code and portability. The presented implementation is shown on development of web application example in order to justify our design choices.

I. INTRODUCTION

Software Product Lines (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 [1]. The well-known concepts in this area are Generative Programming [2], pre-processor definitions, components, Aspect Oriented Programming, Feature-Oriented Programming (FOP) [3], [1], Aspectual Feature C Modules (AFMs) [4] and frames like XVCL [5]. Using SPL helps to increase the software making productivity, by producing it in a way comparable to industrial production.

This paper presents the implementation in Python of the source code generator that is based on the SCT dynamic frames model. The SCT model consists of three basic components: Specification (S), which describes the application characteristics, Configuration (C), which describes the rules for building applications, and Templates (T), which refers to application building blocks. The SCT model is described in detail in paper "Source Code Generator Based on Dynamic Frames", which is currently under reviewing procedure for journal. The third section of this paper describes basics of SCT model. The SCT model is primarily designed for web application development, but there are no constraints to using the SCT model in development of any kind of a source-code, regardless to problem domain and programming language. At first, 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.

The paper is organized as follows: Related work is presented in section 2. The basics of the SCT model are presented in section 3. Section 4 describes implementation of the SCT model in Python, which is followed by one example in section 5. The conclusion is given in section 6.

II. RELATED WORK

Our approach is comparable to Xml-based Variant Configuration Language (XVCL) [6]. Jarzabek's XVCL is a frame mechanism based on Bassett's frames. 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 [7]. 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 model, 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 XVCL. 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 (which is similar to dynamic polymorphism based on virtual methods and mechanism of late binding).

We used Python object features and flexibility as a scripting language in generator implementation. Unlike some other scripting languages such as Perl and PHP, Python is well founded for manipulating complex classes

and is therefore not only suitable for character strings processing. For example, Python lists can contain elements of different types, where any element can easily be replaced by another, regardless of their (possibly incompatible) types.

There are some other projects that use scripting languages in code generation. Some of them are oriented to building new scripting languages, dedicated to making generators, such as Open Promol [8] and CodeWorker [9]. Other projects use existing scripting languages, including Python. For example, Cog transforms program files so that it finds chunks of the Python code embedded in them, executes the Python code, and inserts its output back into the original file [10].

For implementation of our previous generators, based on the Scripting Generator Model [11] that SCT is based on, we used Perl, C++ and Java as implementation languages. Perl, as a scripting language, has flexible data structures (e.g. fields with unlimited number of elements) and possibilities in strings processing. C++ has a full object mechanism that enables defining a generator as a series of generative objects [11]. We used Java as the implementation programming language for building the source code generator, the purpose of which was the dynamic generation of Web services using ontology [12]. Python offers both object-oriented and scripting possibilities, both of which were used for the SCT based generator implementation.

III. SCT MODEL BASICS

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

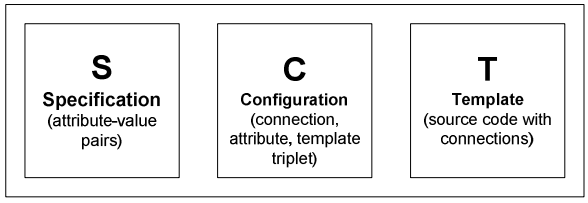

Fig. 1: SCT frame

- 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[1] 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):

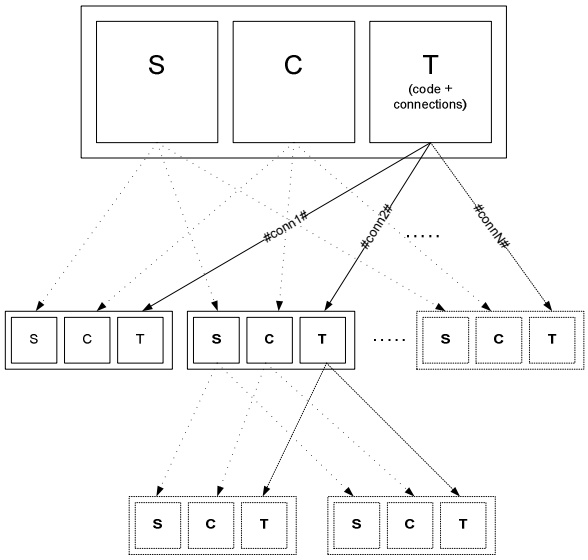

Fig. 2: The generation tree

The each frame produces one fragment of source code in process of program generation. The final program code is built from all fragments of source code by source code generator.

IV. IMPLEMENTATION OF SCT MODEL IN PYTHON

The structure of the Python based SCT generator is defined by these two parts (Fig. 3): *Handler* and SCT class (defining the generator). SCT class enables the generation process, including forming of SCT generation tree, while *Handler* connects generator with the environment and makes generator scalable.

4.1. Handler

The role of Handler is to prepare inputs for the generator (SCT object) and to collect and save generator outputs. Handler defines a new SCT object and initializes it. It also finds each particular output file name and the appropriate part of Specification as well as the appropriate base template from Configuration. After this, Handler invokes the generator and saves the generated code to the target output file (Fig. 3). In Fig 3 are shown part of Configuration and part of Specification by using XML notation as defined in SCT model[1]. The content of Configuration and Specification in Fig 3 is described in more details in next section.

As shown in Fig. 3, Handler prepares inputs for the SCT object by extracting the required parts of Specification and Configuration.

[1] XML schema of SCT frame is available at http://generators.foi.hr/xml_schema.jpg

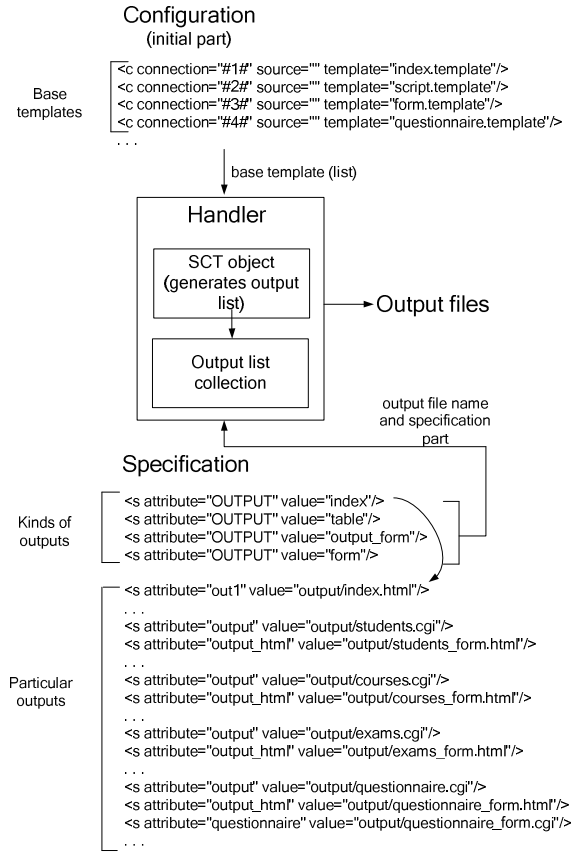

Fig. 3: Functions of *Handler*

4.2. SCT class

The SCT class implements the generator as a recursive structure that starts by initial Specification, Configuration and base template, defined as Python lists:

```
class SCT:
    specification=[]
    configuration=[]
    template=[]
```

Specification list contains attribute-value pairs, loaded from Specification (Fig. 4). Each element of such list is a pair that contains two elements:

- attribute name and
- attribute value

Some attribute names include '+' sign, or even more '+' signs. That means that this attribute is subordinated to previous hierarchical level (with one'+' less, or without '+' sign).

Attribute names are further used in Configuration, which is also in a form of Python list.

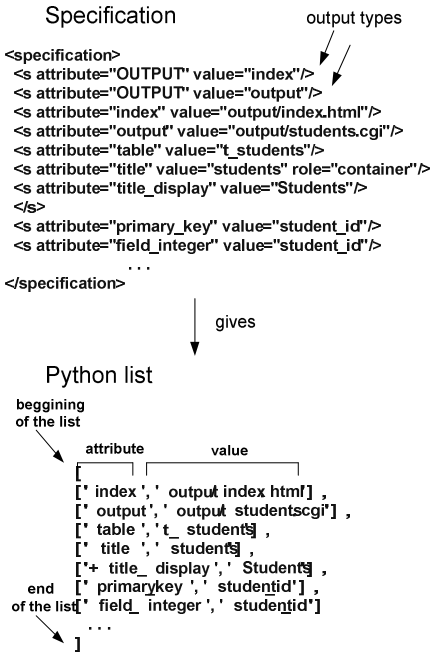

Fig. 4: Specification and Python list

Output types are processed by the function of Handler, and are therefore not included in the Python list.

Configuration list consists of three elements lists (connection, Specification attribute and template), as shown in Fig. 5:

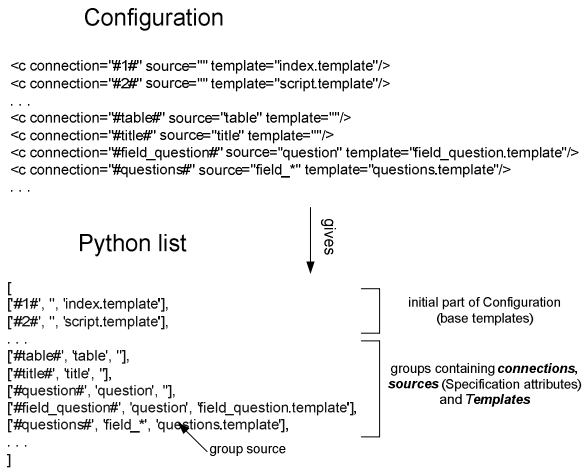

Fig. 5: Configuration and Python list

The order of list elements is unimportant.

Template contains the base template of the generator, also in form of a list, where the program code is separated from connections. An example of template is shown in Fig 6 (connections are in bold).

```
[
'$sql="select $field from ',
'#table#',
' order by ',
'#primary_key#',
'";\n',
'#question_type#',
'\n'
]
```

Fig. 6: An example of template in form of Python list

Connections are to be replaced by attribute values, or by an SCT object (with Python flexibility coming into play), which depends on the appropriate Configuration list element, e.g.:

```
['#primary_key#','primary_key',']
```

- replacement by attribute value

but

```
['#question_type#','question',
'question_type.template']
```

- requires that the connection *#question_type#* is going to be replaced by an SCT object (initialized by the appropriate part of Specification and Configuration), which results in a new template list, e.g. (Fig. 7):

```
[
'$sql="select $field from ',
'students',          <-- value for #table#
' order by ',
'student_id',        <-- value for #primary_key#
'";\n',
<__main__.SCT instance at 0x0181A7D8>,   <-- reference to SCT object
'\n'
]
```

Fig. 7: Template list

The new list contains the SCT object as a whole generator that should generate code for the appropriate connection. Such a change of type (from a string to an SCT object) is easily viable in Python and is one of the main reasons for using that language in generator implementation.

V. EXAMPLE

The example generator[2] deals with web surveys in a form of questionnaires. It produces *html* forms and *cgi* scripts (in Perl) that do the job with a database: database table creating, entering the answers from web form to database table, showing the results of a survey in a form of web pages and similar. The generator itself is made in Python and works in a web environment. The basic model elements, Specification, Configuration and Templates are available in *html* form (contains textual representation of XML frame), as shown in Fig 8.

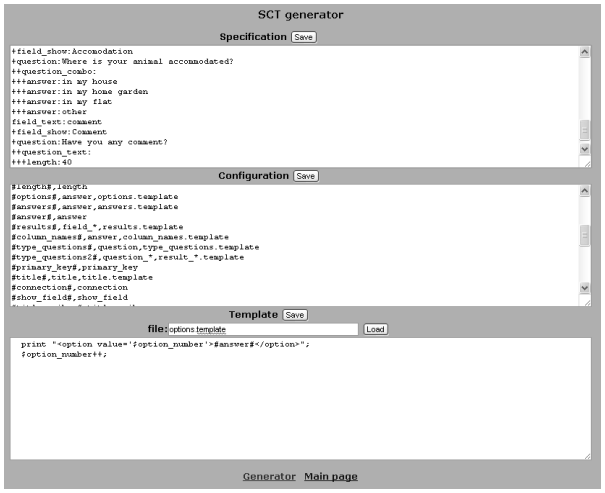

Fig. 8: Basic model elements in *html* form

The generated web application includes index page with links to different operation with questionnaires (could be more than one questionnaires), web questionnaire forms and review of results (graphic and textual). The example application includes questionnaire about pets (Fig. 9).

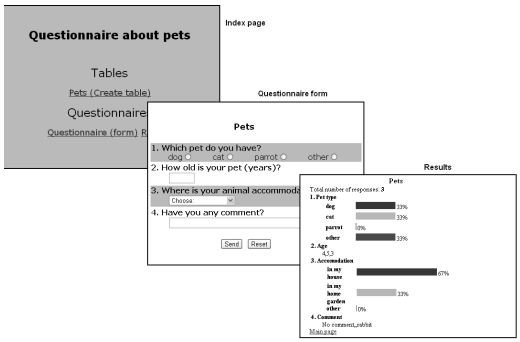

Fig 9: Parts of generated application

5.1. Specification

The Specification in its first part defines names for the kinds of outputs to be generated. Each kind of output is connected to appropriate top-level template in Configuration. In the example generator, kinds of outputs are, as follows (attribute-value pairs; textual representation):

OUTPUT:index
OUTPUT:table
OUTPUT:output_form
OUTPUT:form

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

where *index* refers to the index page (*html*), *table* to Perl scripts for database table creation, *output_form* to *html* form of questionnaire, and *form* to Perl scripts for maintaining questionnaires. The definition of particular question looks as follows (attribute-value pairs; textual representation):

```
field_number:type
+field_show:Pet type
+question:Which pet do you have?
++question_radio:
+++answer:dog
+++answer:cat
+++answer:parrot
+++answer:other
```

Attribute *field_number* defines that database table field is numeric (particular type, like *integer*, depends on appropriate code templete), *field_show* defines what will be shown in results (instead of question), *question_radio* defines the form of question (radio buttons) and attributes *answer* defines possible answers to the question. Signs '+' define hierarchic levels (e.g. *field_show* belongs to *field_number*).

5.2. 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):

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

Configuration	Specification
#1#,,index.template	OUTPUT:index
#2#,,script.template	OUTPUT:table
#3#,,form.template	OUTPUT:output_form
#4#,,questionnaire.template	OUTPUT:form

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:

```
#table#,table
```

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,

```
#links#,title,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:

```
#form_fields#,field_*,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:

```
#fields#,list(field_*)
```

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.

5.3. Templates

Templates are program code fragments that contain connections in '#' signs. For example, Fig. 10 shows the template of a web application index page (HTML):

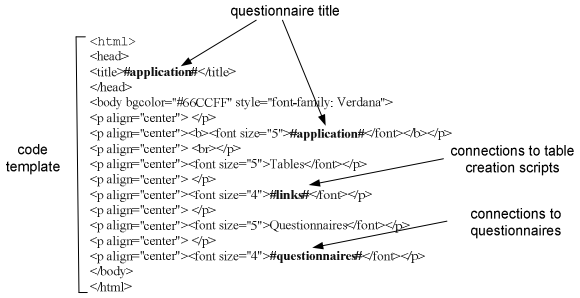

Fig. 10: Example of code template

Each connection has appropriate line in Configuration that define how the connection should be replaced by code, e.g. the Configuration line:

```
#application#,application
```

defines that #application# should be replaced by the value of *application* attribute (in Specification).

```
#links#,title, links.metascript
```

defines that #links# should be replaced by the template *links.metascript*, as many times as *title* occurs in Specification. All connections in *links.metascript* should be processed in the same way (according to their Configuration lines).

VI. CONCLUSION

This paper presents the Python implementation of SCT generator model for defining, building and documenting of a source code generator. The model defines three components: Specification, Configuration, and a set of Templates. These three components together make SCT frames. The model was compared to Jarzabek's XVCL[6].

It is shown that Python is a suitable language for building SCT based generators, because of its flexibility as a scripting language with object-oriented features, including possibility of creating their own classes. In generator implementation, basic model elements, Specification, Configuration and Templates are represented in a form of Python lists. These lists are very flexible data structures that enable mixing elements of different types, including replacing strings by objects, which is usable in program code generation. The presented implementation model is verified by building a generator of web applications that deals with questionnaires.

REFERENCES

[1] M. Rosenmüller, N. Siegmund, G. Saake, S. Apel, "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.

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

[3] C. Prehofer, "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.

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

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

[6] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang, "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.

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

[8] V. Štuikys, R. Damaševičius, G. Ziberkas, "Open PROMOL: An Experimental Language for Target Program Modification," Software Engineering Department, Kaunas University of Technology, Kaunas, Lithuania, 2001., http://soften.ktu.lt/~damarobe/ publications/Vytautas_Stuikys.pdf

[9] C. Lemaire, "CODEWORKER Parsing tool and Code generator - User's guide & Reference manual," http://codeworker.free.fr/CodeWorker.pdf, 2008.

[10] Cog web site. Available at: http://nedbatchelder.com/code/cog/, last accessed 20-12-2010.

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

[12] I. Magdalenić, D. Radošević, Z. Skočir, "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