Implementation Model of Source Code Generator

Ivan Magdalenić, Danijel Radošević, and Dragutin Kermek

***Abstract* - The on demand generation of source code and its execution is essential if computers are expected to play an active role in information discovery and retrieval. This paper presents a model of implementation of a source code generator, whose purpose is to generate source code on demand. The implementation of the source code generator is fully configurable and its adoption to a new application is done by changing the generator configuration and not the generator itself. The advantage of using the source code generator is rapid and automatic development of a family of application once necessary program templates and generator configuration are made. The model of implementation of the source code generator is general and implemented source code generator can be used in different areas. We use a source code generator for dynamic generation of ontology supported Web services for data retrieval and for building of different kind of web application.**

***Index Terms* - source code generator, generative programming, Web service, data retrieval**

I. INTRODUCTION

The emphasis of Semantic Web is on semantic data description, which allows computers to play an active role in information discovery and retrieval. If computers are expected to give an automatic response to a user request, in addition to the semantic description of data it is necessary for computers to have an additional functionality such as on demand generation of source code, including its compilation and execution.

This paper presents an implementation model of a source code generator (IMSCG). Although IMSCG is developed for the purpose of dynamic generation of ontology supported Web services for data retrieval, its definition is general and can be used in different areas. The role of the source code generator within Semantic Web applications is to generate source code for new applications and to enable computers to respond dynamically depending on a semantically defined user's request. The architecture and model of dynamic generation of ontology supported Web services for data retrieval is already presented in [1][2], albeit without a detailed description of the implementation model of the source code generator. We have also use IMSCG for building a web application [3].

Manuscript received July 11, 2010; revised April 16, and May 31, 2011.

This work has been partially supported by Ministry of Science and Technology, Croatia, in 2010.

Authors are with the Faculty of Organization and Informatics, Varaždin, University of Zagreb, Zagreb, Croatia (email:fivan.magdalenic, danijel. radosevic, dragutin.kermek@foi.hr).

IMSCG uses previously developed scripting model of application generator (SMG; [4]) for model description, but its implementation uses new configuration that is separated from generator's code, meaning that generator is now fully configurable.

The main intention of this paper is to provide a implementation model of the source code generator that is fully configurable and can be easily adopted for many problem domains. The definition of IMSCG is independent of the programming language and can be implemented in different programming languages. However, the implementation of IMSCG is easier in programming languages which support recursion. The advantage of using a source code generator thus defined is rapid and automatic development of a family of application once necessary program templates and generator configuration are made. Since the source code generator is fully configurable, its adoption to a new application is done by changing the generator configuration and not the generator itself. The verification of the presented IMSCG is done by its implementation in Java for the purpose of dynamic generation of Web services for data retrieval.

The paper is organized as follows: Related work is presented in section 2. The implementation model of the source code generator with an illustrative example is presented in section 3. Section 4 describes the verification of the model described in section 3. The conclusion is given in section 5.

II. RELATED WORK

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. Model Driven Development and Software Product Lines (SPL) are two cases in point [5]. 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 [6]. The well-known concepts in this area are Generative Programming [7], pre-processor definitions, components, Aspect Oriented Programming, Feature-Oriented Programming (FOP) [8],[6], Aspectual Feature C Modules (AFMs) [9] and frames like XVCL [10]. Using SPL helps to increase the software making productivity, by producing it in a way comparable to industrial production. By using concepts of Generative Programming (GP), SPL can be fully automated, which is an important characteristic of IMSCG.

FOP treats software features as fundamental units of abstraction and composition. IMSCG uses application specification that defines which program code templates will be used in the final output. This is similar to FOP, but at a lower lever of definition of features.

The way in which pre-processor definitions are used in making problem-domain adjustments is presented in [6]. In IMSCG problem-domain adjustments are made by changing the configuration of the source code generator, which does not affect the existing program code templates.

IMSCG is oriented to working with code-fragment-sized components. The same approach is used in [11]. Other GP based projects, like Uniframe [12], [13], avoid descending to code-fragment-sized components.

Some approaches are based on manipulation or generation of programs within the language, which requires a language with metalanguage capabilities. Languages like `C (Poletto 1999) [14] and DynJava [15], provide such facilities. C++ provides a solution with template metaprogramming [7], where generated programs are expressed as parameterized types, and code is produced by a compiler through inlining [6]. IMSCG avoids inlining specific to a particular programming language, which enables the generation of source code in any programming language.

The main specific difference between IMSCG and other template engines, such as Velocity [16], is in moving the instruction for handling templates from program code templates into a separate file that is used for configuration of the source code generator.

Our approach in building the source code generator has some similarities with frames-based approaches, especially XVCL [10]. Both models are hierarchic, implemented as a tree-like multi-level structure of code templates (XML frames in XVCL [10]). Also, the basic principle in both models is that lower template levels adapt their superposed templates during the process of generation. This is opposite to standard inheritance in object-oriented programming, where lower-level classes inherit the members of superposed classes. However, the differences between our approach and that used in XVCL are even more important: our specification is separated from templates (while in XVCL specification among frames is shared); there is no need to specify semantic elements like class or variable names in specification, because the generator works as a macro mechanism that can produce all kinds of textual outputs (e.g. documentation as well as the program code); finally, all templates are reusable, i.e. can be used at different parts and hierarchic levels while the XVCL frames contain the information about their superposed frames.

III. IMPLEMENTATION MODEL OF THE SOURCE CODE GENERATOR

The main components of the source code generator are presented in Fig. 1. Inputs to the source code generator are program code templates, application specification, and configuration of the source code generator. The output of the source code generator is a string representation of the program code.

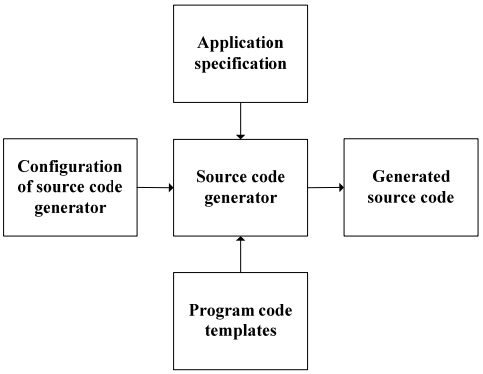

Fig. 1. Model components of source code generator

A. Program code templates

Program code templates are code fragments stored in separate files. Program code templates contain the program source code and replacing marks. Replacing marks are user defined names separated with special characters. Replacing marks are enclosed with a special character #, e.g. #replacingMark1#. Examples of five program code templates are shown in Fig. 2. These program code templates are made for the purpose of retrieving data from different data sources. The replacing marks in program code templates in Fig. 2 are italicized.

As shown in Fig. 2, replacing marks are used for different purposes: for type and variable names (e.g. *#argumetType# #argument#*), method names (e.g. *#methodName#*), pieces of data (e.g. *#username# and #password#*) but also for representing larger pieces of code (e.g. *#dataSource#* should be replaced by source code for retrieving data from a different kind of data source and *#filters#* by code to implement data filters) where lower-level templates are used.

```
public void #methodName#(#arguments#){                          method
   #dataSource#                                                 .code
}

#argumentType# #argument#,                                      argument
                                                                .code

 {
  databaseAccessObject.Xml d=
    new databaseAccessObject.Xml();
  d.setPath("#source#");
  #filters#                                                     dataSource
  while (dx.hasNext(){                                          xml.code
    System.out.print("Model="+modelName+
      ":price="+d.get("#response#"));
  }
 }

 {
  databaseAccessObject.Xml d=
    new databaseAccessObject.Xml();
  d.setCredentials("#username#","#password#");
  d.setTableName("#source#");
  #filters#                                                     dataSource
  while (dx.hasNext(){                                          oracle.code
    System.out.print("Model="+modelName+
      ":price="+d.get("#response#"));
  }
 }

d.setFilter("#filterName#", #filterValue#);                     filter.code
```

Fig. 2. Examples of program code templates

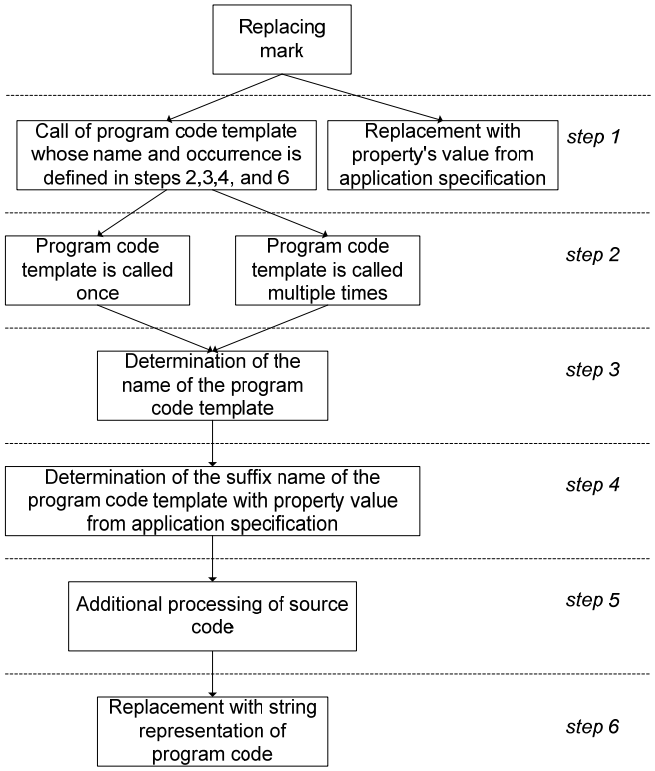

Fig. 5. Steps in replacing marks replacement

For each replacing mark in step 1 the source code generator searches for its definition in the configuration of the source code generator.

A replacing mark is replaced with a property value from the application specification if the definition of replacing marks has syntax:

*replacingMark*_**specification**=*propertyName*

where *replacingMark* is the name of the replacing mark in the program code template, **_specification** is the keyword for replacement with the property value from the application specification, and *propertyName* is the property name from the application specification whose value is used to replace the replacing mark. If there are more properties with the same name, the source code generator takes the value form the property that occurs first in the list.

When this step is completed, the source code generator handles the next replacing mark. If a replacing mark has to be replaced with a program code template, the source code generator performs steps 2, 3, 4, 5 and 6 from Fig. 5.

Step 2 offers the implementation of **for statement**. For each defined property occurrence in the application specification, a program code template is called, which name is formed in steps 3 and 4. The syntax is following:

*replacingMark*_**foreach**=*propertyName*

where *replacingMark* is the name of the replacing mark in the program code template, **_foreach** is the keyword for implementation of *for statement*, and *propertyName* is the property name from the application specification.

In step 3, the source code generator reads the name of the program code template from the configuration of the source code generator. The definition in the configuration of the source has the following syntax:

*replacingMark*_**name**=*value*

where *replacingMark* is the name of the replacing mark in the program code template, **_name** is the keyword for the definition of the program code template name, and *value* is the actual name of the program code template that will be called. This definition is mandatory for replacing marks which are changed with a program code template. All program code templates have the extension *.code*.

Step 4 is optional and enables the forming of the program code template name by concatenating the name formed in step 3 with values from application specification properties. The syntax is following:

*replacingMark*_**virtual_1**=*propertyName_1* *(1)*
*replacingMark*_**virtual_2**=*propertyName_2* *(2)*
…
*replacingMark*_**virtual_n**=*propertyName_n* *(3)*

where *replacingMark* is the name of the replacing mark in the program code template, **_virtual** is the keyword for the definition of the suffix of the program code template name, **_n** is the sequence number of the suffix, and *propertyName_n* is the property name from the application specification whose value is concatenated with the program code name from step 3.

This way of name formation offers a new functionality and introduces program logic into the process of calling program code templates. Since the name of a program code template is formed with values of properties, the generation process is controlled by the application specification. The names of called program code templates are defined dynamically and are not known before the process of source code generation. This process can be used to implement **if statement** by choosing an appropriate program code template depending on the values of properties from the application specification, which is an important feature of IMSCG.

Step 5 offers additional source code processing, when all replacing marks are replaced by another program code template or by property values. An example of such processing is the removal of the last comma in some enumerations. This step is used for handling exceptions that cannot be solved by appropriate program code templates. Functions called in this step have to be implemented within the source code generator. The syntax is following:

*replacingMark*_**function**=*value*

where *replacingMark* is the name of the replacing mark in the program code template, **_function** is the keyword for calling a function, and *value* is the function name implemented in the source code generator.

source code generator. Numbered field lines show which step from Fig. 5 is performed.

Fig. 9 shows how the replacing mark #dataSource# is replaced with program code templates **dataSourcexml.code** and **dataSourceoracle.code** (steps 2, 3, 4, and 6 from Fig. 5).

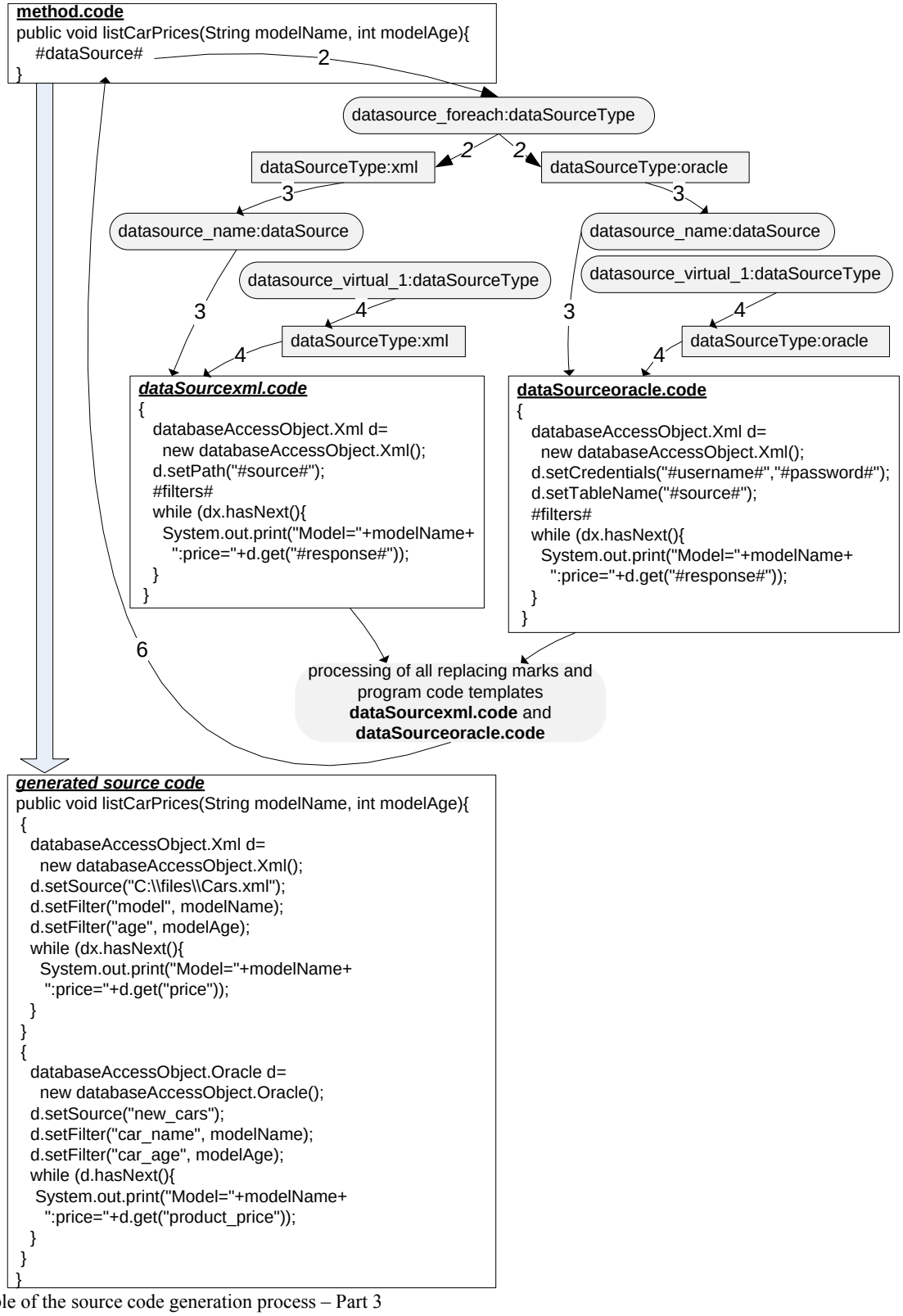

Fig. 9. Example of the source code generation process – Part 3

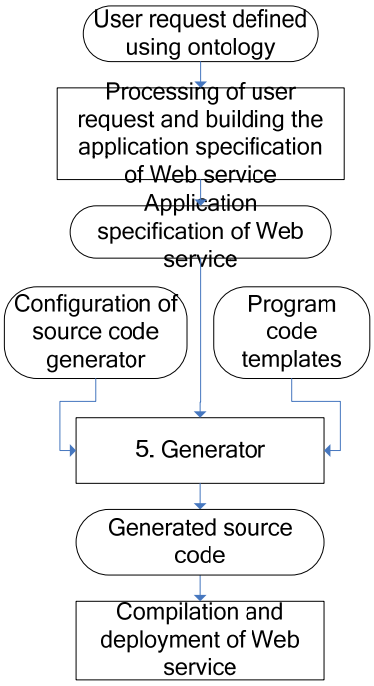

Fig. 11. Model of dynamic generation of ontology supported Web services for data retrieval

This IMSCG is implemented in the programming language Java. The main part of the model implementation is recursive function, which handles replacing marks in programming code templates as described in the subsection "Configuration of the source code generator". The generated Web services are used for data retrieval from data sources of the following types: XML, MS Excel and RDBMS Oracle. We have accomplished that specification of one Web service has in average 23,8 times less definition elements with regard to definition elements in generated source code. Definition elements in source code are keywords, variables, and constants.

For the purpose of building web application, we made an implementation of IMSCG in Python. We use Python's flexibility as a scripting language, together with the object-oriented possibilities. The base for implementing generators is usage of Python lists. It's important that Python lists can contain elements of different and even non-compatible types. The lists contain configuration, specification, and code templates.

V. CONCLUSION

This paper presents an implementation model of the source code generator. Its purpose is to enable easy implementation of the source code generator in any programming language and to use it for generation of a complete application in different problem domains.

The presented model has many similarities to the approaches listed in the "Related work" section. It is a template engine like Velocity, which works with code-fragment-sized components and its principle of specification of application is similar to Feature-Oriented Programming. What distinguishes it from other approaches is the extraction of template engine logic from program code templates. Namely, program code templates in the presented model contain only one type of replacing marks. The replacing logic is stored in a separate configuration file and is loaded by the source code generator. The main advantage of this approach is greater reusability of program code templates, which can be used in different problem domains by changing the configuration of source code generator. Another advantage is the configurable source code generator, where replacing logic is not hardcoded and, therefore, can easily be changed.

The practical applicability of the presented model is tested on the generation of Web services for data retrieval and different web application.

In our future work we plan to focus on problems of checking consistency of the model implementation.

REFERENCES

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

[2] I. Magdalenić, B. Vrdoljak, Z. Skočir, "Towards Dynamic Web Service Generation on Demand," Proceedings of the International Conference on Software, Telecommunications and Computer Networks 2006, September 2006.

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

[4] D. Radošević, T. Orehovački, M. Konecki, „WEB oriented applications generator development through reingineering process," DAAAM International Scientific Book 2007, DAAAM International, Vienna, Austria 2007.

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

[6] 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.

[7] K. Czarnecki, U. Eisenecker, *Generative programming: methods, tools and applications*, Addison-Wesley, 2000.

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

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

[10] H. Zhang, S. Jarzabek, „XVCL: a mechanism for handling variants in software product lines," Science of Computer