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.

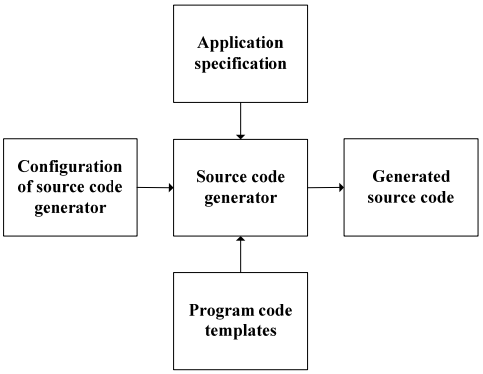

Fig. 1. Model components of source code generator

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

B. Application specification

The application specification is a list of property-value pairs. The application specification defines one application from a family of possible applications defined by configuration of the source code generator. The application specification and configuration of the source code generator define the process of source code generation by choosing the appropriate program code template and by providing concrete values for replacing marks in program code templates.

An example of application specification is shown in Fig. 3. It specifies a method named "listCarPrices". The method has two input arguments: "modelName" of type "String" and "modelAge" of type "int". It retrieves prices of cars from two data sources. The first data source is an XML file stored in "C:\files\Cars.xml". The second data source is a table with the name "new_cars" in the Oracle database. The part of specification which defines access to the Oracle database has more properties compared to the data source of xml type. In this example, the application specification specifies the argument name because it is used later as a value in property filterValue. Other types of application can generate argument names automatically.

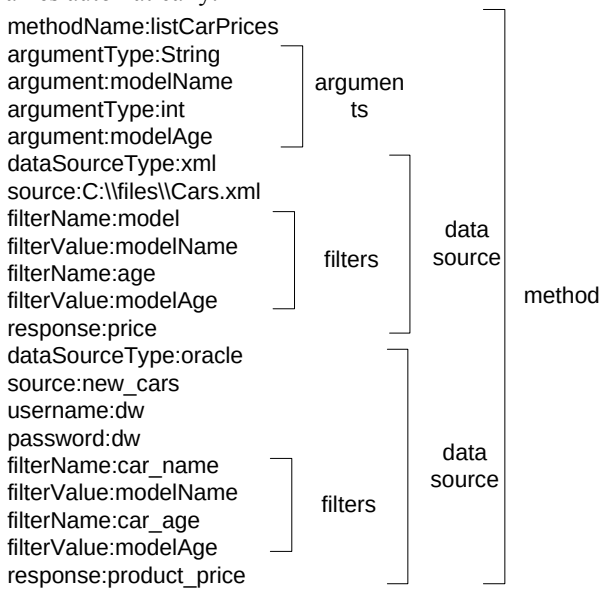

Fig. 3. Example of application specification

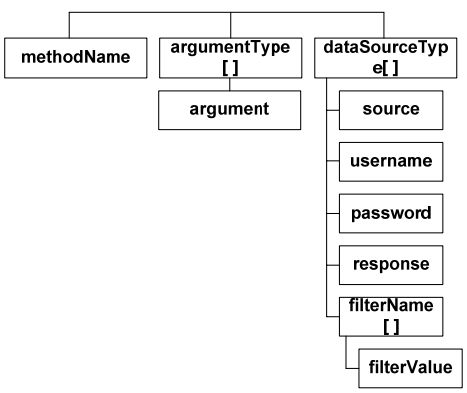

Fig. 4. Example of application specification diagram

The application specification has a hierarchic structure. If a certain property is related to another property, it can be listed only if the related property comes before in the list. The hierarchy structure can be presented in many ways. This paper uses a simple structure diagram for presentation of a hierarchy structure of application specification (for a similar approach see: Limbourg and Kochs [17]). An example of such a structure diagram is shown in Fig. 4 for an example of application specification presented in Fig. 3.

Rectangles in Fig. 4 represent property names. Properties marked by [] are containers for lower-level properties. The usage of property values is defined in configuration of the source code generator.

C. Configuration of source code generator

The configuration of the source code generator defines what source code generator have to do with replacing marks in program code templates. The main difference between the approach in this paper and other approaches is that the generator's executable code does not have to be changed for particular problem domains and that the entire logic of replacement of program code templates is stored in the configuration of the source code generator. The configuration of the source code generator defines a family of similar applications.

The essential part of the procedure of generating source code is the replacement of replacing marks with a value from its specification or with another program code template.

In the former case it is necessary to clearly define which property value from the application specification is going to be used at a certain point since each property can occur multiple times (for example, methods may have more arguments with different data types).

In the latter case it is necessary to determine which program code template has to be called and used instead of the replacing mark. In other words, it is necessary to create a specific way to manage the selection of one program template code between all the available ones. The conventional approach to this problem is to store the logic of selection of a certain program code template into the executable code generator or into program code templates. Subsequent changes are difficult because they require recompiling of the source code generator executable code. Therefore, if the selection logic is stored in program code templates and changes are required, they have to be changed or new program code templates have to be made. This paper proposes the creation of the source code generator that has selection logic stored in a separate configuration file. The source code generator performs the steps shown in Fig. 5, which are described simultaneously with the parameters describing the configuration file of the source code generator. After the source code generator loads the program code template, it performs a search for replacing marks and performs steps for each replacing mark presented in Fig. 5. An illustrative example of all the steps is shown in the next subsection.

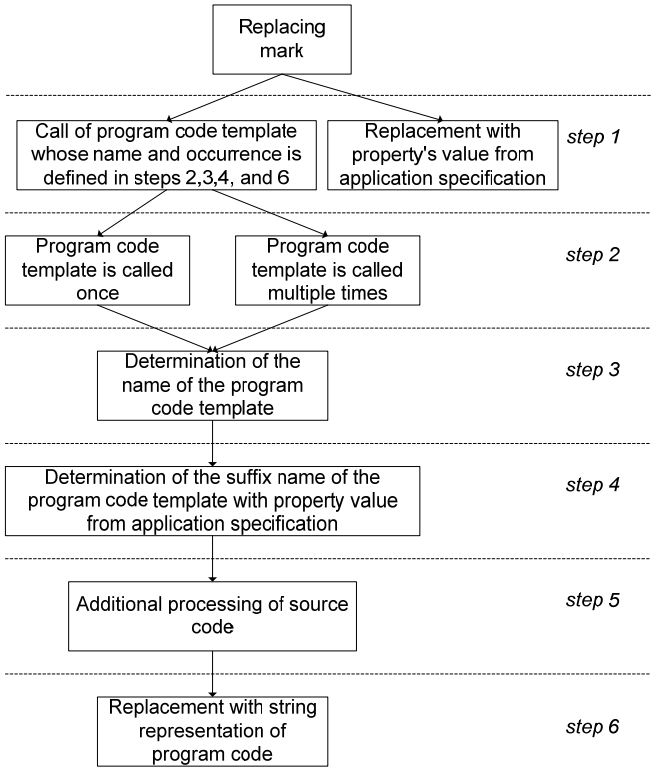

Fig. 5. Steps in replacing marks replacement

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.

In step 6 each replacing mark is replaced with string representation of source code from called program code template. Once it has been decided which program code template is going to replace which replacing mark, the source code generator first searches for replacing marks in that program code template and performs the steps from Fig. 5. The process of handling replacing marks is recursive and the program code template that is called last, returns the string representation of source code etc.

Fig. 6 shows the configuration of the source code generator for generation of a method whose functionality is retrieving prices from different data sources and printing them on the screen. The meaning of the configuration of the source code generator presented in Fig. 6 is as follows. The replacing mark *#arguments#* is replaced with a program code template with the name *argument.code* as many times as the property *argumentType* occurs in the application specification (as shown in Fig. 8). For the last called program code template a function is called which will remove the last comma in the string representation of source code.

```
arguments_foreach:argumentType
arguments_name:argument
arguments_funtion:comma                    - configuration for replacing mark #arguments#

datasource_foreach:dataSourceType
datasource_name:dataSource
datasource_virtual_1:dataSourceType        - configuration for replacing mark #datasource#

filters_foreach:filterName
filters_name:filter                        - configuration for replacing mark #filters#

methodName_specification:methodName
argumentType_specification:argumentType
argument_specification:argument
source_specification:source
response_specification:response
username_specification:username
password_specification:password
filterName_specification:filterName
filterValue_specification:filterValue      - configuration for replacing marks with values from application specification
```

Fig. 6. Example of source code generator configuration

The replacing mark *#datasource#* is replaced with a program code template with the name *dataSourcePROPERTYVALUE.code*, where *PROPERTYVALUE* is the value of property *dataSourceType*. This program code template is called as many times as the property *dataSourceType* occurs in the application specification (as shown in Fig. 9). The replacing mark *#filters#* is replaced with a program code template with the name *filter.code* as many times as the property *filterName* occurs in the application specification.

D. Illustrative example of source code generation

An example of the process of source code generation using the presented model is shown in Fig. 7, 8, and 9. In this example program templates from Fig. 2, application specification from Fig. 3, and configuration of source code generator from Fig. 6 are used.

Fig. 7 shows how the replacing mark #methodName# is replaced with a value from the application specification. The field line marked with 1 refers to step 1 in Fig. 5.

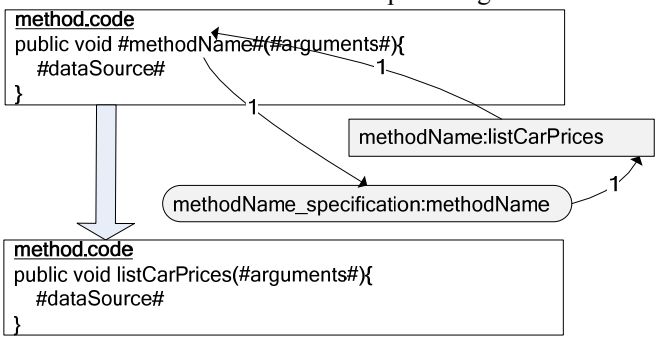

Fig. 7. Example of the source code generation process – Part 1

Fig. 8 shows how the replacing mark #arguments# is replaced twice with the program code template **argument.code**, because property *argumentType* occurs twice in the application specification (steps 2, 3, 5 and 6 from Fig. 5).

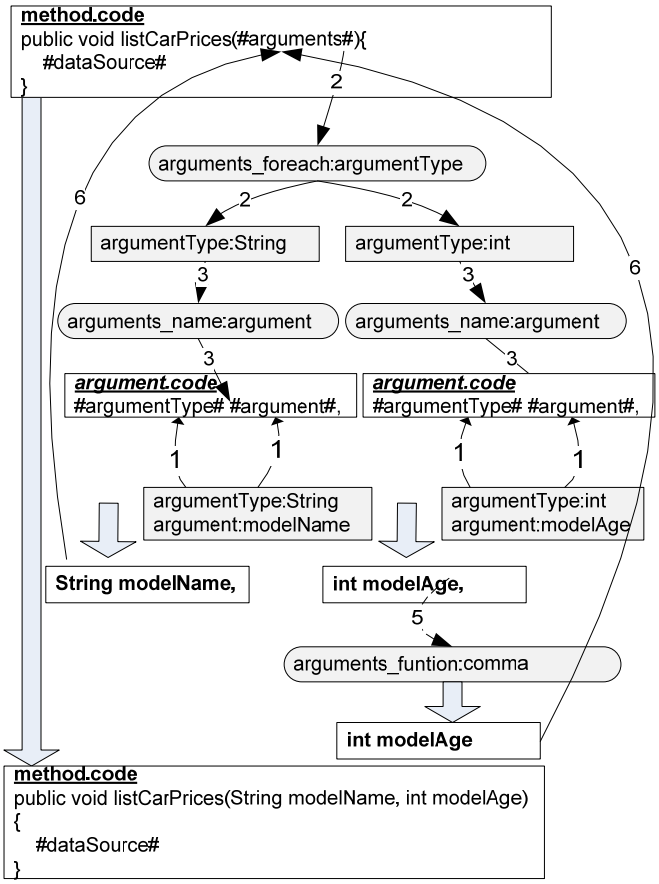

Fig. 8. Example of the source code generation process – Part 2

In the second program code template argument.code is removed comma by calling a function that is built into the source code generator and defined in the configuration of the

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).

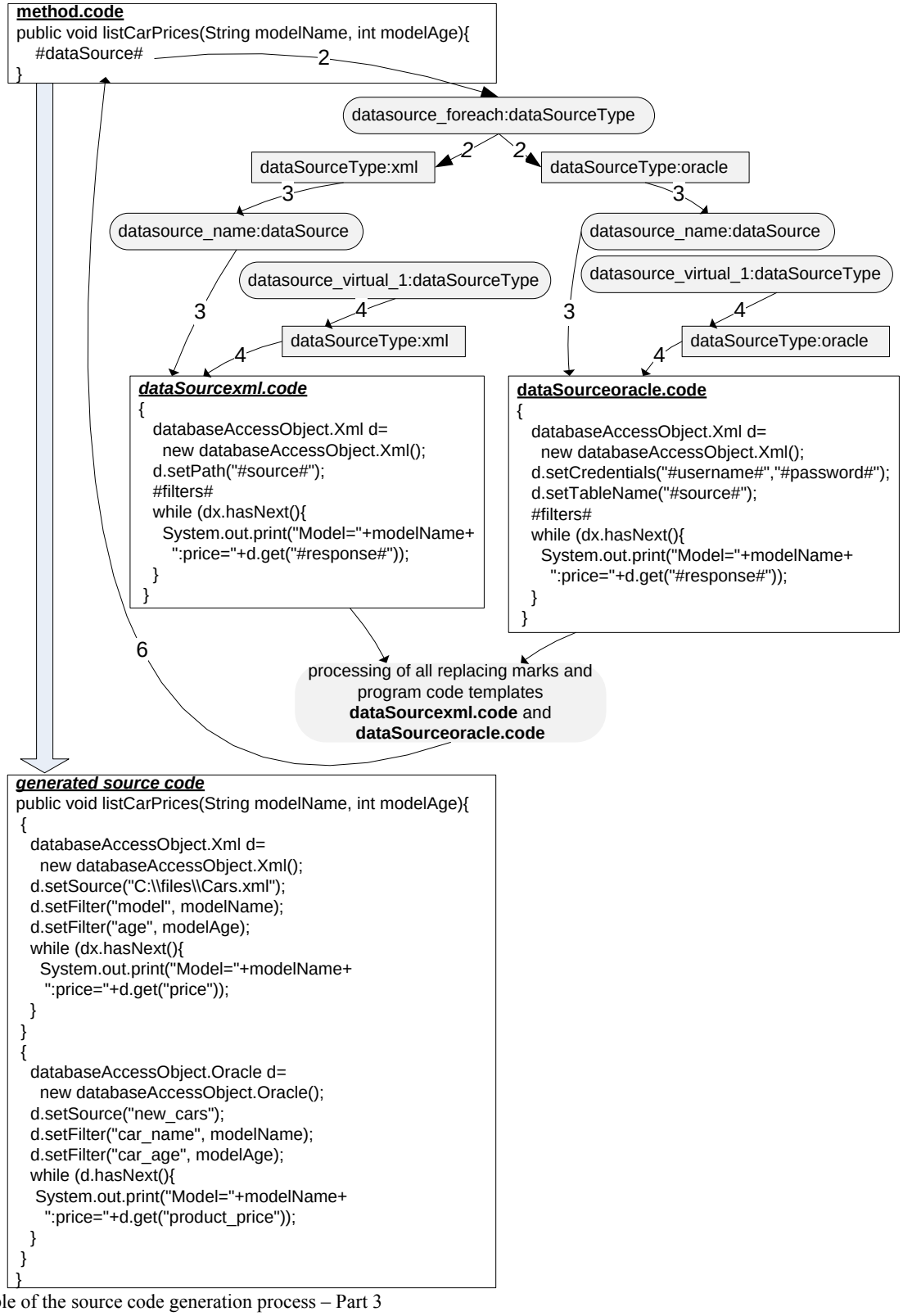

Fig. 9. Example of the source code generation process – Part 3

The configuration line *dataSource_name:dataSource* defines that the beginning of the name of the program code template is *dataSource*. The configuration line *dataSource_virtual_1:dataSourceType* defines that the value of property with the name dataSourceType is used to concatenate with the already defined beginning of the name of the program code template. In the first case that value is *xml*, and in the second case *oracle*. The processing of replacing marks in program code templates dataSourcexml.code and dataSourceoracle.code is not described in detail. Fig. 9 shows the final result of the source code generation process.

Filled rectangles in Fig. 7, 8, and 9 represent pieces of program specification, filled rectangles with round corners represent the configuration of the source code generator, while rectangles which contain replacing marks in '#' signs represent program code templates. The generated source code is generated by the replacement of replacing marks with lower-level templates or specification values by using rules described in the previous section.

E. Generative application development

Generative application development is the process of parallel development of generators, together with target applications (Czarnecki and Eisenecker [7]). It is implemented in IMSCG, as shown in Fig. 10:

The starting point of generative application development with IMSCG is the application prototype. Program code templates are extracted from the prototype, where features are replaced by replacing marks. The starting specification should define the application prototype. The configuration defines connections between the specification and templates. Therefore the starting generator should replicate the prototype application. After that the generative application development defines levels of generator refinement, which are connected to developers' roles:

- the *domain engineer's* role is responsible for the development of generator, including its configuration and testing;
- the *software programmer's* role is responsible for application prototypes and development of program code templates;
- the *user's role* is responsible for defining features of target applications in form of application specification and testing of generated applications.

It is important that the user's role should deal with the application specification, which has to be separated from the program code level.

Generative application development faces to issues due to model consistency and generated programs correctness. The main issues are as follows:

- *Mistakes in specification*. Using undefined attributes, unadmitted attribute values or missing the attribute hierarchy. Some could be avoided by generating system, but some remains for the testing phase. These mistakes are typical for the user's role.
- *Improper code templates.* Syntax/logical errors in code templates remains for the testing phase, while using of undefined replacing marks could be detected by the generating system. Software programmer's role is responsible for this kind of issues.
- *Improper configuration.* Syntax mistakes like referencing of non-existing templates or usage of connections that do not appear in Templates in configuration could be checked by the generating system. On the other hand, configuration is covered by the domain engineer's role, so logic mistakes at that level could lead to useless generated code.

The issues could be reduced by the appropriate generative application development process, including testing phase on different levels of generators and generated applications.

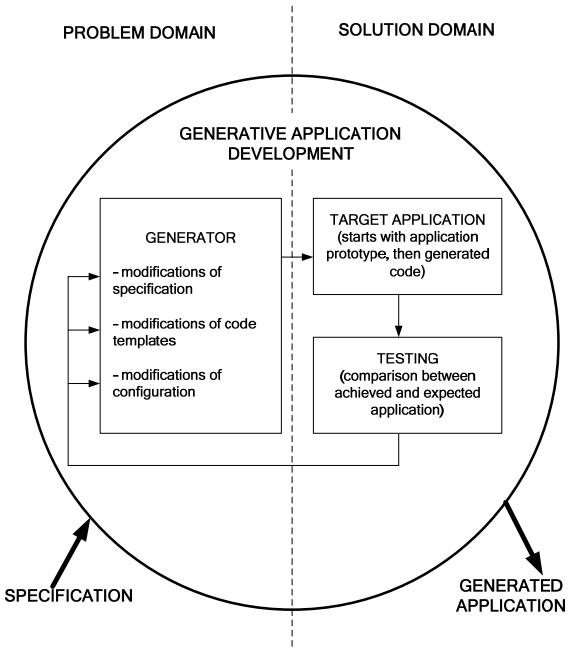

Fig. 10. Generative application development

IV. IMSCG VERIFICATION

The implementation model of the source code generator presented in this paper is verified within implementation of the application for dynamic generation of ontology supported Web services for data retrieval and different web applications. The architecture and the model of dynamic generation of ontology supported Web services to retrieve data are presented in [1] and [2]. The basic idea is presented in Fig. 11.

User defines request where semantic meaning of data are represented using ontology classes or ontology properties. The next step is processing of user request and building of application specification of Web service, which is input to source code generator. The configuration of source code generator defines how to produce source code from program code templates and application specification. The generated source code is compiled, deployed to web container, and make available to use.

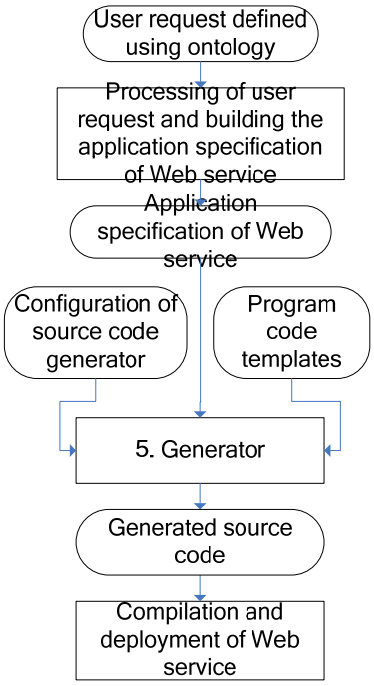

Fig. 11. Model of dynamic generation of ontology supported Web services for data retrieval

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

Programming, Volume 53, Issue 3 (December 2004) Pages: 381 – 407

[11] M. L. Griss, „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, 2001.

[12] A.M. Olson, R.R. Raje, B.R. Bryant, C.C. Burt, M. Auguston, „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, 2005.

[13] Uniframe web site (http://www.cs.iupui.edu/uniFrame/)

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

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

[16] Velocity web site (http://velocity.apache.org/).

[17] P. Limbourg, H.D. Kochs, „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, Pages 815-828, 2008.

Ivan Magdalenić, PhD is an assistant at the University of Zagreb, Faculty of Organization and Informatics in Varaždin. His research interests are in e-Business, Web technology, Semantic Web technology and generative programming. He has been involved in several projects of e-business adoption in Croatia. He is a member of National council for e-Business.

Danijel Radošević, PhD, is an associate professor at University of Zagreb, Faculty of Organization and Informatics. He teaches at different programming courses at undergraduate studies and professional studies. His Ph.D. thesis was focused on usage of scripting languages in generative programming, while current research deals with programming languages, generative programming and educational software.

Dragutin Kermek joined the University of Zagreb Faculty of Organization and Informatics in 1992 as a teaching assistant. He holds a Ph.D. (1999), M.Sc. (1992) and B.Sc. (1986) in Information Science from The University of Zagreb Faculty of Organization and Informatics Varaždin. He is an Associate Professor from 2007, and has served at Faculty of Organization and Informatics as a Vice dean for Academic affairs from academic year 2005/2006 to 2010/2011. He has taught courses: Advanced Web technologies and Services, Web Design and Programming, Design Patterns, Operating Systems, Chosen chapters from Component architectures and technologies, E-learning Systems, Chosen chapters on instructional design and mentoring in e-education. He has been project manager of several e-learning projects related to developing, programming, and implementation of e-Learning platforms and web based information system solution, using electronic media to provide trainings and e-Learning platforms expertise.