DEFINITION OF A SEMANTIC PLATAFORM FOR AUTOMATED CODE GENERATION BASED ON UML CLASS DIAGRAMS AND DSL SEMANTIC ANNOTATIONS

DEFINICIÓN DE UNA PLATAFORMA SEMÁNTICA PARA LA GENERACIÓN AUTOMÁTICA DE CÓDIGO BASADA EN DIAGRAMAS DE CLASES UML Y ANOTACIONES SEMÁNTICAS EN UN DSL

ANDRÉS MUÑETÓN

M.Sc, Universidad Pontificia Bolivariana, Medellín, andres.muneton@upb.edu.co

CARLOS ZAPATA

Ph.D. Universidad Nacional de Colombia, Medellín, cmzapata@unal.edu.co

Received for review July 6 th, 2011, accepted November 12th, 2011, final version January, 11th, 2012

ABSTRACT: In this paper, we propose a semantic service platform for implementing the steps of a semantic- and model-driven architecture (MDA)-based method for automated code generation. The code generation is achieved by semantically relating operations in unified modeling language (UML) class diagrams with implemented operations. The relationship among operations is achieved by finding implemented operations with the same post-condition of the operation under implementation. The resultant code is a sequence of invocations to the implemented operations which, acting as a whole, achieve the post-condition of the operation under implementation. Semantics is specified by means of a domain-specific language (DSL), also defined in this paper. Services of the platform and the method are shown in execution in a case study.

KEYWORDS: Code generation, automation, MDA, semantic, platform, software engineering.

RESUMEN: En este trabajo se propone una plataforma semántica de servicios que implementan los pasos de un método para la generación automática de código. El método se basa en información semántica y en MDA (model-driven architecture). La generación de código se logra relacionando semánticamente operaciones en diagramas de clases en UML (unified modeling language) con operaciones implementadas. La relación entre operaciones se hace consultando operaciones implementadas que tengan la misma postcondición de la operación bajo implementación. El código resultante es una secuencia de invocaciones a operaciones implementadas que, en conjunto, alcancen la postcondición de la operación bajo implementación. La semántica se especifica mediante un DSL (domain-specific language), también definido en este artículo. Los servicios de la plataforma y el método se prueban mediante un caso de estudio.

PALABRAS CLAVE: Generación de código, automatización, MDA, semántica, plataforma, ingeniería de software.

1. INTRODUCTION

Automation in software development is a very important topic of research and application. By using automated activities, the development process makes the team more agile so that they may spend time in more crucial activities besides manual processes. In other words, automation supports the engineering approach to software development. Some examples of automated activities in software development are: generating source codes, testing (e.g., unit and user acceptance tests [1]), integrating [2], and delivering [3].

Current code generation proposals exhibit some problems: They still offer partial results or they are too formal to be fully implemented and useful for the industry. Three main sets of proposals have been identified: code generation from graph models, code generation from formal representations, and code generation from a mixture of graph and textual expressions. Purely graph-based approaches [4–5] have limitations due to the expressiveness flaws of graph languages. On the other hand, formal proposals use formal languages to specify every aspect of the system and, consequently, generate the entire code [6]. Also, formal languages are hard to

learn and apply. Finally, some researchers have opted for a combination of models and textual specifications [7–8]. However, these proposals are too close to the platform or suitable only for a very specific domain.

Zapata and Muñetón [9] have proposed a basic method for code generation from the UML meta-model instances and semantic annotations. With this method, the code is intended to be generated by using a set of transformation rules defined by Muñetón et al. [10] and database single-semantic operations like "insert," "update," and "delete."

In this paper, we propose a semantic platform for automated code generation. This platform improves the previous method of Zapata and Muñetón [9] in supporting more complex operations by using a model-driven architecture (MDA)-based structure and including steps that are more formal.

This paper is structured as follows: In the next section, we give an overview of proposals and platforms for code generation. Then, in Sections 3 and 4 we present the improved method of code generation, and a semantic platform that implements this method, respectively. Section 5 shows an example of the application of the method in the platform, and finally, conclusions and future work are stated.

2. RELATED WORK

From a state-of-the-art review in code generation, three main sets of proposals have been identified: code generation from graph models, code generation from formal representations, and code generation from a mixture of graph and textual expressions—both in formal representations or using domain-specific languages (DSLs). This section presents some approaches for each type. At the end of the section, we will discuss the main issues concerning these approaches.

There are several proposals for accomplishing automatic code generation by using graph transformations, most of them under the MDA scope. Cooper et al. [4] allow code generation from models to aspects in AspectJ, a java implementation of aspect-oriented programming (AOP). The transformation among models is accomplished by means of Extensible Markup Language (XML) specifications and meta-models of XML and AspectJ. The code is generated from the XML specifications and the aspects are controlled in the system by throwing and handling exceptions. Nassar et al. [5] propose a method for code generation by merging use-case-based view point models, logic, component, and deployment. The models are stereotyped according to elements of VUML (View-based Unified Modeling Language) and, then, transformed into code by using predefined rules specified in the ATLAS Transformation Language (ATL). The aim of the generated code is to manage the different views of the system, excluding business logic.

In contrast to graph transformation approaches, some proposals use a formal language as a source model for code generation. They use formal languages due to their well-defined semantic. This is the case of PADL2Java [6], which defines transformation rules to generate code in Java from specifications in an algebraic formal language called PADL. The system is completely specified on PADL, easing the process for generating both structural and behavioral codes.

Finally, some researchers have opted for a combination of models and textual specifications, and the benefits of both. Fang [7] combines modeling patterns and action semantic with MDA to create applications for a specific platform, called EJB. Patterns and platform are expressed with UML and action semantic representations. The latter is based on the UML meta-model. Although the use of meta-models could allow for the extension of the method to other platforms, the strong link between patterns and the platform could make this step difficult. Also, in some cases the code obtained from patterns is structural and not behavioral. Sánchez et al. [8] shows a graphic DSL for the home automation domain. The structure of the DSL is composed of so-called *functional units,* common functionalities on the domain (e.g., light power on/off or lighting level regulation associated to services like dimmers and timers.) A program with DSL consists of a sequence of services and its actions. The code generated is fully executable due to the defined behavior exhibited by every functional unit, which acts as a code template. The DSL has an internal XML representation, the main source of the transformation rules. The proposal is domain-platform independent because its structure is flexible enough to allow for the implementation of new services and functional units in several devices and platforms.

The use of formal specifications for modeling the code generation process has some drawbacks.

Designers and developers are accustomed to use graph models, especially in UML—the *de facto* standard for modeling—and, additionally, formal languages could be harder to learn and apply. Also, the need for writing the entire code of the system in a source model is a negative factor, no matter whether the code was written using a formal language or the benefits of it. On the other hand, purely graphical approaches have limitations too, due to the flaws of graph languages. Despite its evolution to languages closer to code—UML, for example—it is still difficult to express any aspect of a system with enough level of detail for code generation. The recent approaches show that MDA is still in a stage of development, especially in core aspects such as the platform independent model (PIM)-to-platform specific model (PSM) transformation.

The principal flaw of the aforementioned proposals is related to behavioral code—the code of the methods—which is not generated or predefined as a template. Formal methods have exceptions to this fact, because they can generate the entire code of a system, but starting from a complete source code in a formal language. DSLs act as alternatives, but the approaches analyzed are either too close to the platform or they are suitable only for a very specific domain.

The following sections show a novel approach for code generation which combines UML graphs and semantic annotations. This improvement allows for one to generate structural and behavioral codes and uses a DSL for making the annotations easy enough to learn and understand for developers, designers, and business stakeholders.

3. METHOD FOR THE CODE GENERATION BY MEANS OF SEMANTIC RELATIONSHIPS

The proposed method for code generation is based on MDA, as it considers the PIM and PSM models. Also, the Meyer design by contract [11] is included, especially the use of pre- and post-conditions to represent the semantic of operations. The pre-conditions of an operation are the restrictions for executing it, while post-conditions are the properties of the state of the system immediately after the execution of the operation.

In addition to the aforementioned properties, in this method we introduce semantic annotations for model-to-model and model-to-platform relationships, specified in a DSL.

The steps of the method are: (i) PIM model creation; (ii) solution-independent semantics (SIS) annotation to PIM; (iii) solution-specific semantics (SES) annotation to PIM; (iv) relationship settlement between the semantically annotated PIM and the platform; and (v) code generation.

The first step of the method is the creation of the PIM model. Since the model-to-code transformation rules act on instances of the UML meta-model, PIM must be a well-formed class diagram. Next, in step two, the intention of each operation is expressed with a solution independent semantic (SIS)—similar to the programming-by-intention initiative—in which the developer starts writing sentences for expressing the objective he/she expects to reach, instead of specifying how it could be reached [12]. Then, the intention is the "what" and the solution is the "how" of the operation. In contrast, SES of the PIM+SES construction, or step three, expresses a concrete solution for the operations of the models; in this step, the model is still independent of the implementation platform. Both steps two and three act independently of each other. Consequently, they could be executed in any order. In the next step, the model semantically specified is related to the platform for allowing code generation. This step requires a target platform with semantic specifications to be accomplished, and the rules defined by Muñetón et al. [10]. The connection between PIM and PSM is a variation of MDA because no transformation rules are needed, but a relationship between models. Section 6 illustrates this with a case study.

A formal representation of the method is:

cg: M_{SEM}, $P_{SEM} \rightarrow C$

where "$M_{SEM} = (M_{SIS}, M_{SES})$", is the semantic specified model, "P_{SEM}" is the platform with its semantic specification, "C" is the set of the generated code, and "bg" is the behavioral code generation function. Given this, "bg" can be represented as:

bg: {mop1, mop2, mop3, …, mopn}, {pop1, pop2, pop3, …, popm} → {Iop1, Iop2, Iop3, …, Iopn}

where "mop"i is an operation of the semantic model, "pop"i an operation of the platform, and "Iop"i is the set of implementations of operation *i*. "Iop"i is defined as a finite sequence of invocations to implemented operations of domain or platform:

Iopi=(invoke(op1), invoke(op2), invoke(op3),…, invoke(opn)

Semantic annotations of "Msem" and "Psem" are specified by using a DSL, whose Extended Backus-Naur Form (EBNF) is given below:

semantic = 'SIS' | 'SES', 'to', identifier,':','precondition=', empty | assertion, 'postcondition=',assertion;

assertion = subject, space, predicate, space, object;

subject = instantiation | identifier;

predicate = 'in' | 'not-in' | 'type-of' ;

object = 'Collection' | dbtable;

instantiation = identifier, '(', [arguments], '):', type;

identifier = letter, {letter | capitalizedLetter};

type = capitalizedLetter, {letter | capitalizedLetter};

arguments = identifier, {',', identifier};

dbtable = 'DB(table=', identifier, ')';

letter = 'a'..'z';

capitalizedLetter = 'A'..'Z';

space = ' ';

empty = '';

A semantic specification has two assertions: a "pre-condition" and a "post-condition." An "assertion" is composed by three elements: "subject," "predicate," and "object." In this paper, the domain of the DSL is restricted by the predicates: "in," "not-in," and "type-of." Also, we use operations to add and remove elements on collections and to insert and update records of a database table.

4. SEMANTIC COMPUTING PLATFORM FOR AUTOMATIC CODE GENERATION

In this section we present the platform architecture for allowing the implementation of the method introduced in the previous section. The structure is based on the semantic computing architecture [13] that considers five core layers and two transversal ones: security and management.

Each layer in the platform has a concrete objective reached by services, which are intended to have a general purpose in the original proposal. The data to be processed in the platform are analyzed and converted to semantics in the information analysis layer. These semantics are related to the content in the next layer of semantic integration. With the output of this layer, the services of the semantic services layer can perform activities; e.g., generate code. A set of services could be related to create a more powerful service by means of the service integration layer. Finally, there are services in a semantic interface layer for allowing a natural interaction with the final user.

Figure 1 shows the semantic computing architecture with concrete services for code generation: SIS analyzer, SES analyzer, integrator, and code generator. Furthermore, two repositories of code were added: a code corpus and the API. In this paper, we are only focusing on the central components of the platform, excluding the management and security layers.

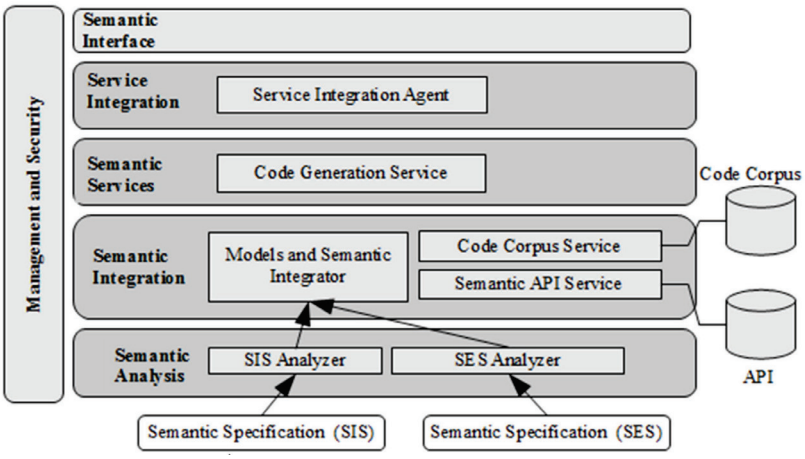

Figure 1. Semantic platform with services for automated code generation, adapted from [13]

The representation of the DSL sentences can be easily understood by the final user, because the services of the semantic analysis layer make a syntactic analysis, without using an interface of interaction. This is why there are no services in the semantic interface layer.

The services of the semantic integration layer allow for the implementation of steps (ii), (iii), and (iv) of the code generation method. The models and the semantic integrator service match models with semantics and other models. This is the case of the PIM+SIS and PIM+SES relationships. The platform-specific constructions are achieved by the code corpus service and the semantic API service, which are related to code repositories.

There are two types of code generation services: domain operation and platform operation code generators. The domain operation code generator produces the code of the signature of the user model operations and the code of the body. (This task is carried out with the help of the platform operation code generators). For each platform operation there is a code generator.

Finally, the service integration layer coordinates the interaction of the platform services.

5. CASE STUDY

In this section we show an implementation of the method presented in Section 4 by means of a case study based on the process of an invoice. An invoice has one or more details, and each detail has a product and a quantity. An invoice has three operations—addDetail, deleteDetail, and save—which add a detail to the invoice details, delete a detail from the details of the invoice, and save the invoice in database, respectively.

5.1. PIM Model Creation

According to the method, the first step is to create a PIM model. Figure 2 shows a class diagram that acts as the PIM of the invoice case study.

5.2. PIM and SIS Relationship

Next, a semantic-independent solution is assigned to the PIM in order to obtain the PIM+SIS model. This is specified by using the DSL as seen in Fig. 2.

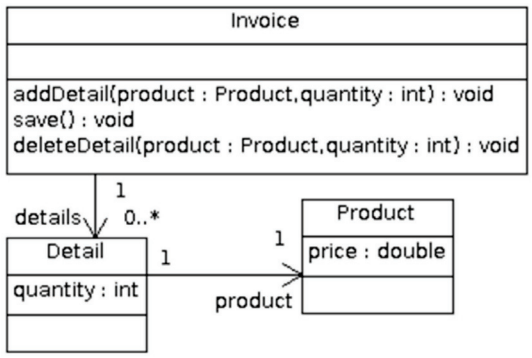

Figure 2. PIM diagram for the case study

SIS to addDetail:

precondition=detail(product, quantity):Detail not in details

postcondition=detail(product, quantity) in details

The previous SIS specifies the intention of addDetail: "to add a detail to the details."

5.3. PIM and SES Relationship

In this step, a semantic specific to the solution is used to specify that "details" is intended to be a collection.

SES to ddDetail:

precondition=

postcondition=details typeof Collection

SES relates the operation with a solution, not with the platform. Then, collection is not a concrete platform class, but a grouping of data items.

5.4. Platform Relationship

The combination of SIS and SES for the addDetail operation is the semantic: "addDetail must add a detail, which is an object composed by a product and a quantity, in a collection called details."

Once this semantic is obtained, the next step is focused on finding an implemented operation whose contract could be related with the contract of "addDetail." In this case, the operation is the "add" of the Java platform. The post-condition of "add" is "e in :Collection", an abstract representation for adding elements into a specific collection. The relationship between semantics

is more clear if "e" is replaced by detail(product, quantity), and :Collection by details.

SES to add:

precondition=

postcondition=e in :Collection

5.5. Code Generation

Finally, the code is generated from the previous relationships.

void addDetail(Product product, int quantity){

Detail detail=new Detail(product, quantity);

details.add(detail);

}

The code is generated according to the transformation rules defined by Zapata and Muñetón [9]. The body of "addDetail" is an invocation of "add" operation. This sentence is generated by a code generator service associated with the "add" operation.

The code for saving an invoice and removing a detail is generated in a similar way.

The semantic specification of "deteleDetail" and the generated code are:

SIS to removeDetail:

precondition=detail(product,quantity):Detail in details

postcondition=detail(product,quantity):Detail notin details

SES to removeDetails:

precondition=

postcondition=details typeof Collection

The full semantic of "deleteDetail" is "deleteDetail must remove a detail of the collection details." The "remove" operation of the java.util.Collection interface has a contract that matches the deleteDetail semantic:

SES to remove:

precondition=e in :Collection

postcondition=e notin :Collection

The generated code is:

public void deleteDetail(Product product, int quantity){

Detail detail=new Detail(product,quantity);

details.remove(detail);

}

The method for code generation checks the existence of the identifiers generated in the code. In the case of "addDetail" and "deleteDetail," the "detail" object has to be created because it does not exist as an argument of the operation.

Nowadays it is uncommon to write the entire code to interact with the database. The main reason is the existence of a lot of frameworks and tools that hide this code from the programmer by automatically generating the source code. This is the case of hibernate or the Java Persistence API. However, in this work we decide to show, due to pragmatic aspects, the semantic specification of the "save" operation.

SIS to save:

precondition=invoice:Invoice notin invoices

postcondition=invoice:Invoice in invoices

SES to save:

precondition=

postcondition=invoices typeof DB(table=invoices)

The full semantic for the "save" operation is: "save must insert a detail, and object composed by a product and a quantity, in a database table called invoices".

The platform "execute" operation has a contract that match the save contract.

SES to execute:

precondition=

postcondition=e in DB(table=es)

The code generated for save is:

void save(){

Connection con=DriverManager.getConnection();

PreparedStatement ps=con.preparedStatement();

ps.execute();

}

The code generation is still incomplete. Parameters of some operations, such as getConnection() and preparedStatatement(), are not obtained. Consequently, the save operation is not executable and it may be considered for future work.

The semantic specifications of the operations invoked in the body of "save" are:

SES to preparedStatement:

precondition=:PreparedStatement is void

postcondition=:PreparedStatement isnot void

SES to getConnection:

precondition=:Connection is void

postcondition=:Connection isnot void

6. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a set of services, structured on a semantic platform, to generate the code of operations from UML class diagrams.

A method for code generation was also proposed. The method is based on MDA and Meyer's Design by Contract. Instead of diagram transformations, such as MDA, the method relates to models by using semantic annotations. The semantic annotations have two types: solution-independent and solution-dependent. The former allows for one to specify the intention of the operation, avoiding technical aspects, which are indicated with the solution specific semantic.

To facilitate the semantic specification of the operations, a DSL was defined. The DSL is easy to learn and apply, allowing for its use by technical and non-technical members of the development team.

As future work, the method for code generation must be improved by considering the arguments of the platform operations in the generated code. The bodies of such operations require the use of conditionals and loops, and the addition of error handling.

REFERENCES

[1] Crispin, L. and Gregory, J., Agile Testing: A practical Guides for Testers and Agile Teams, Addison-Wesley Professional, Boston, MA, 2009.

[2] Duvall, P., Matyas, S. and Glover, A., Continuous Integration: Improving Software Quality and Reducing Risk, Addison-Wesley Professional, Boston, MA, 2007.

[3] Humble, J. and Farley, D., Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley Professional; 1 edition. Boston, MA, 2010.

[4] Bennett, J., Cooper, K. and Dai, L., Aspect-oriented model-driven skeleton code generation: A graph-based transformation approach, Science of Computer Programming, 75, pp. 689-725, 2010.

[5] Nassar, M., Anwar, A., Ebersold, S., Elasri, B., Coulette, B. and Kriouile, A., Code Generation in VUML Profile: A Model Driven Approach, IEEE/ACS International Conference on Computer Systems and Applications, pp. 412 – 419, 2009.

[6] Bontà, E. and Bernardo, M., PADL2Java: A Java Code Generator for Process Algebraic Architectural Descriptions, IEEE/ACS International Conference on Computer Systems and Applications, pp. 412 – 419, 2009.

[7] Cheng, f. MDA Implementation Based on Patterns and Action Semantics, Third International Conference on Information and Computing, 25-28, 2010.

[8] Sánchez, P., Jiménez, M., Rosique, F., Álvarez, B. and Iborra, A., A framework for developing home automation systems: From requirements to code, The Journal of Systems and Software. Vol. 84, pp. 1008–1021, 2011.

[9] Zapata, C.M. and Muñetón, A., Generación del cuerpo de los métodos a partir de la semántica de las operaciones del diagrama de clases, Revista Ingeniería e Investigación. Vol 28, No. 3, pp. 58-63. 2008.

[10] Muñetón, A., Zapata, C.M. and Arango, F., Reglas para la Generación Automática de Código Definidas Sobre Metamodelos Simplificados de los Diagramas de Clases, Secuencias y Máquina de Estados de UML 2.0, Dyna, Vol. 74, No. 153, pp. 267-283, 2007.

[11] Meyer, B., Object-Oriented Software Construction, Prentice Hall, Santa Barbara, California, 1997.

[12] Martin, R., Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall, Boston, MA, 2008.

[13] Sheu. P., Semantic Computing, Wiley-IEEE Press, 2010.