I.J. Information Technology and Computer Science, **2017, 12, 11-18**
Published Online December 2017 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijitcs.2017.12.02

Multi-Platform Code Generation Supported by Domain-Specific Modeling

Gábor Kövesdán and László Lengyel
Department of Automation and Applied Informatics, Budapest University of Technology and Economics, Budapest, Hungary
E-mail: {gabor.kovesdan, lengyel}@aut.bme.hu

Received: 01 August 2017; Accepted: 26 August 2017; Published: 08 December 2017

Abstract—Code generation is widely used to make software development more efficient and less prone to human errors. A significant use case of code generation is processing of Domain-Specific Languages (DSLs) and Domain-Specific Models (DSMs). Sometimes, it is desired to generate semantically equivalent or similar functionality to different languages to better support multiple platforms and achieve better reuse in the tooling. For example, it is convenient if a single tool supports code generating from a DSM to either Java or C#. There has been relevant research on using modeling and model transformations for code generation to multiple platforms. The Model-Driven Architecture (MDA) inherently supports multi-platform code generation based on models. Nevertheless, the MDA standard is a high-level general framework that includes standards, notions and principles but does not specify more concrete methods or workflows about their efficient adoption. Our research focuses on the efficient and practically usable application of MDA principles to generate multi-platform code. This paper reports on our results on multi-platform code generation and the difficulties that we are about to addressed in future research. The approach and the challenges presented in the paper are useful for tool developers, such as developers of DSLs, who generates code for several platforms.

Index Terms—Domain-Specific Modeling, Model Transformation, Code Generation.

I. INTRODUCTION

Code generation is a powerful instrument for making software development easier and faster. It has already gained wider adoption in software engineering. Multi-platform code generation is especially demanded because it promotes reuse of models and tools, keeping the development cost and effort low. The Model-Driven Architecture (MDA) [1] maintained by the Object Management Group (OMG) inherently supports multi-platform code generation. Despite the existence of current research and advances in MDA, it still has not gained wider adoption [2]. Reasons include that MDA is idealistically based on a forward engineering approach, making it inflexible. It also requires a deep understanding of the huge amount of related standards or a tool that makes the use of these standards easier. Furthermore, parts of these standards are still not clearly defined, such as the semantics of UML, which is still a subject of research under the term Executable UML [3]. Authors of this paper also consider MDA a high-level general framework rather than a concrete architecture or a method. MDA categorizes models as Platform-Independent Models (PIMs) or Platform-Specific Models (PSMs) and aims to produce PSMs and then executable code from the PIMs provided by developers through a series of model transformations. There is a model transformation chain for each target platform that produces the corresponding PSM. However, nearly any system that generates code from models may fit into this schema. This is why we consider that MDA is an overly general high-level framework that does not guarantee the success of the resulting system in itself. Apart from this, MDA is a general-purpose technology, it aspires modeling a whole system.

As opposed to MDA, Domain-Specific Modeling [4] cannot be used to describe general problems but it aims to efficiently solve problems that belong to a specific problem domain. Because of the domain-specific nature, Domain-Specific Models (DSMs) work with the notions of the problem domain. These are more concrete notions that are more meaningful for domain experts than the notions used in general-purpose modeling. Therefore, DSMs are also more concise and are better understood and contributed to by domain experts, who know the domain but are not necessarily skilled in modeling and computer programming. The advantages of DSMs are further detailed in a large set of books and papers [4][5]. To visualize the information contained in a DSM, usually a Domain-Specific Language (DSL) is used. A DSL may be a textual DSL, which resembles a General-Purpose Language (GPL) but has a syntax that is more suitable for expressing the domain; or the DSL may be a visual DSL that represents the model with an expressive graphical format. The DSM may be interpreted and processed directly as well. However, another common approach is to generate code from the DSM in a GPL. As opposed to general-purpose approaches such as MDA, the code generated from a DSM is not a whole system but a well-defined part of a system that is easy to express with a

DSL, such as business logic from a specific domain. This code is later integrated with manually written components or with code generated from DSMs covering other problem domains that occur in the system. These components together make up the whole software.

The goal of our research is to find an efficient approach for multi-platform code generation that fits into the MDA point of view but is more concretely described to be reusable and does not have the other difficulties of the general MDA approach. We are about to provide a pattern how to do effectively multi-platform code generation. More specifically, we focus on the practical modeling of program logic that can easily be transformed to the concrete syntax. In a full MDA approach, this is the end of the transformation chain that produces executable code in GPLs. We have developed a metamodel that can express general imperative strongly and statically typed object-oriented program code. This metamodel is based on the fact that the semantics of these languages are similar. For example, they deal with class definitions, member variables, method definitions, variable assignments, method calls, and further components that can be handled in a formal way. We use this metamodel to capture the generated code in a way that is as free of language-dependent elements as possible. Once the initial model is transformed to an instance of this metamodel, it can be easily used for code generation. The way the initial model is created and transformed is beyond the scope of this paper. In this paper, we explain our approach that we successfully applied with DSM. It does not aspire to generate complete systems but simpler modules that are later integrated into a larger system. The approach does not require a deep understanding of MDA and can be applied with arbitrary modeling and model transformation tools, without having to comply with the standards suggested by MDA. We believe that our approach explained in the paper supports tool developers in developing software that leverages multi-platform code generation.

The rest of the paper is organized as follows. Section 2 lists related work. Section 3 describes an application of DSM, where multi-platform code generation is required. This example helps understanding better the motivation behind multi-platform code generation and the context in which it may be used. Section 4 presents the approach used for multi-platform code generation in the tool described in the previous section. Section 5 explains the challenges and difficulties that were met in the implementation of the tool. We describe three major difficulties with examples and we provide solutions for them that we applied in our implementation. Moreover, we list further potential solutions that are subject of future research. Section 6 concludes the paper and summarizes the results.

II. Related Work

Related work encompasses mainly two groups of papers. The first set is related to MDA and its general approach. These pieces of work contribute to the rich set of standards and principles, such as Executable UML [3]. Nevertheless, as explained in the introduction, these approaches are too general and warrant further elaboration for the guarantee of success. The present paper aims to provide a more concrete approach that is easily applicable.

The second set includes other concrete languages and tools that try to increase the efficiency of modeling and code generation. ThingML [6] is a tool and a modeling language suggested in place of UML. Interaction Flow Modeling Language (IFML) [7] is another modeling language that aspires making modeling and multi-platform code generation more efficient, especially regarding user interfaces and flow of interaction. The approach explained in the paper is different from these in that it does not focus on how the initial model is created but on the

WOLD [8] is a wizard for generating forms for data manipulation in databases for different platforms. The WL++ [9] tool aims to generate multi-platform mobile clients to RESTful backends. The approach is based on generating code that dispatches platform-dependent tasks to a middleware, PhoneGap, that has a uniform API under several mobile platforms. These tools are different from our approach because they concentrate on specific domains and they generate related code. The method presented in the paper is different because it concentrates on the representation of the code that will be generated. Provided that a model transformation is implemented that transforms the input model into this representation, our approach can be used for any domain.

An earlier work [10] of the authors of the paper is also related because it describes an earlier version of the ProtoKit tool described in Section 3.

III. A Motivating Example on Multi-Platform Code Generation

Nowadays, there are several high-level communication standards that allow for network communication between two pieces of software. One group of these technologies consists of object-oriented remoting standards, like Common Object Request Broker Architecture (CORBA) [11] or Java's Remote Method Invocation (RMI) [12]. The other kind of commonly used technologies includes variants of Web Services, namely, the Simple Object Access Protocol (SOAP) [13] and RESTful Web Services [14]. Despite the availability of these mechanisms, still numerous software vendors decide to develop a lightweight binary application-level protocol that has a lower network footprint and does not require depending on resource-intensive libraries and application servers. We have not found a domain-specific language with code generator that allowed for the modeling of message structure. Existing tools focus more on communication states and interactions [15][16]. Nevertheless, in cloud-enabled applications, the message structure is more relevant. First, these systems do not maintain a permanent connection and their messaging is often limited to notifications and request-response messages. Secondly, lower level protocols hide the establishment and the termination of connections. Because of these factors, the development of cloud messaging primarily consists of determining the message structure and developing the supporting code. Using binary messaging is more challenging to implement than relying on commonly supported formats, such as XML or JSON, that have extensive support in third-party libraries. However, this is the most concise form and thus it generates less network footprint and it is faster to parse. The messaging logic is required for both the client application and the cloud server. If they do not run on the same platform, the supporting code must be developed twice. A DSL and code generation techniques can remedy these difficulties. A code generator can be constructed that uses the model of the message structure and generates the supporting classes and the boilerplate code, even for multiple platforms, if necessary. Such a tool facilitates development and can ensure that the implementations in different languages are consistent. Furthermore, the supporting classes are not that trivial to develop as we would initially imagine. Binary messages often have fields that are not so easy to map to member variables of classes. For example, an integer value may be of diverse lengths, while in GPLs, there are only a fixed number of different integer types. Sometimes, the length of these types is unambiguously defined, such as in Java, sometimes it is platform-dependent, such as in C. This problem must be addressed when protocol messages are mapped to supporting classes in the code generator. Another similar difficulty is using bitfields. To spare with bandwidth, it is common to split one or several bytes into fields of less than eight bits. Such bitfields should practically be accessed with getter/setter methods in the generated classes as if they were regular member variables. At the same time, they require a suitable representation that can easily be serialized and deserialized according to the protocol specification.

In the first version of our ProtoKit tool, we implemented generating Java classes, whereas, as we described in our motivations, the concept supports well generating code in several languages. We have realized that it is quite prone to errors to generate the same thing in several languages with the kind of templates we had in the first version because it is difficult to maintain the templates of different languages in sync. The different versions of the generated code can easily become inconsistent, leading to inconsistent behavior in the application, where the generated code is used. Our new approach for multi-platform code generation that is explained in the next section, successfully solved this problem in the second version of ProtoKit.

IV. The Suggested Code Generation Approach

The second version of the ProtoKit tool is continuously evolving. At the moment, we support generating C++ and Java classes. In this version, we used two separate models. The semantic model describes the message structure as defined in the ProtoKit definition language. This is then transformed with a Model-to-Model (M2M) transformation into another model, which will be referred to as output model. This is an instance of our metamodel that we designed for multiplatform code generation. This model describes elements of general-purpose imperative strongly and statically typed object-oriented languages, such as class, member, method, parameters, return value, assignments, statements, and further elements. Despite programming languages being semantically different in smaller details, languages from this kind share a large amount of commonalities. This makes our approach transparent, efficient and reusable. Since the output model semantically resembles the generated code, it is quite trivial to generate code from it and the approach makes it possible to use a single transformation chain until a very late phase of code generation. In the case of ProtoKit, we only used a single M2M transformation. Although the outcome of this transformation is still a model, the resulting model can be considered as a program that has semantics but not materialized in the concrete syntax of Java or C++. This means that this single M2M transformation is the main place where code generation takes places. Therefore, extensions and bugfixes in the generated code almost always need to be applied only once to the M2M transformation, and the

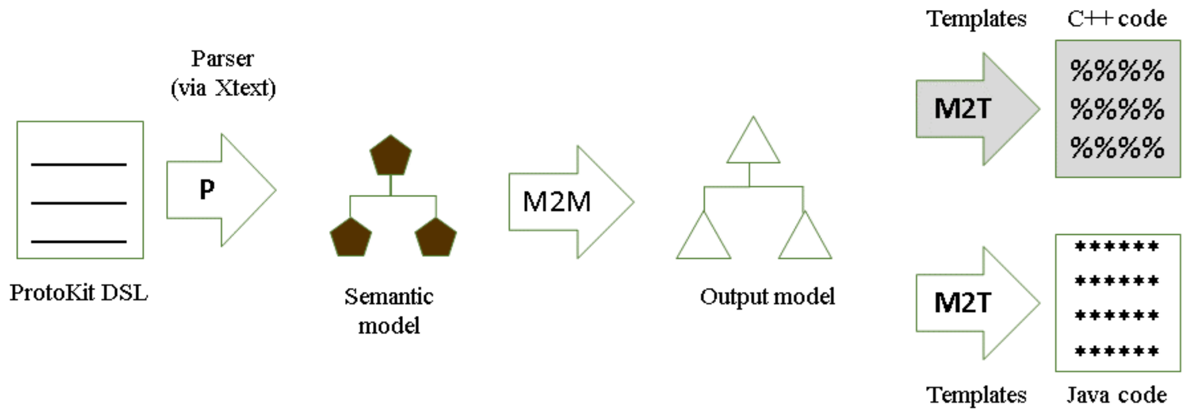

Fig.1. The architecture of ProtoKit 2.

effects of these changes are automatically propagated to the generated classes in both languages. Through this reuse of transformation logic, our approach highly supports multi-platform code generation. The architecture of ProtoKit 2 is depicted in Figure 1.

The metamodel we used for the output model is too large to be represented as a whole in a diagram, therefore we include a subset of it. Basically, it is an object-oriented representation of object-oriented code. For example, it defines packages that are composed of classes. Classes have names and may aggregate member variables and methods. Methods may have return values and a list of parameters and body statements. Statements may be assignments, conditional branches, loops etc. Figure 2 depicts a subset of the metamodel.

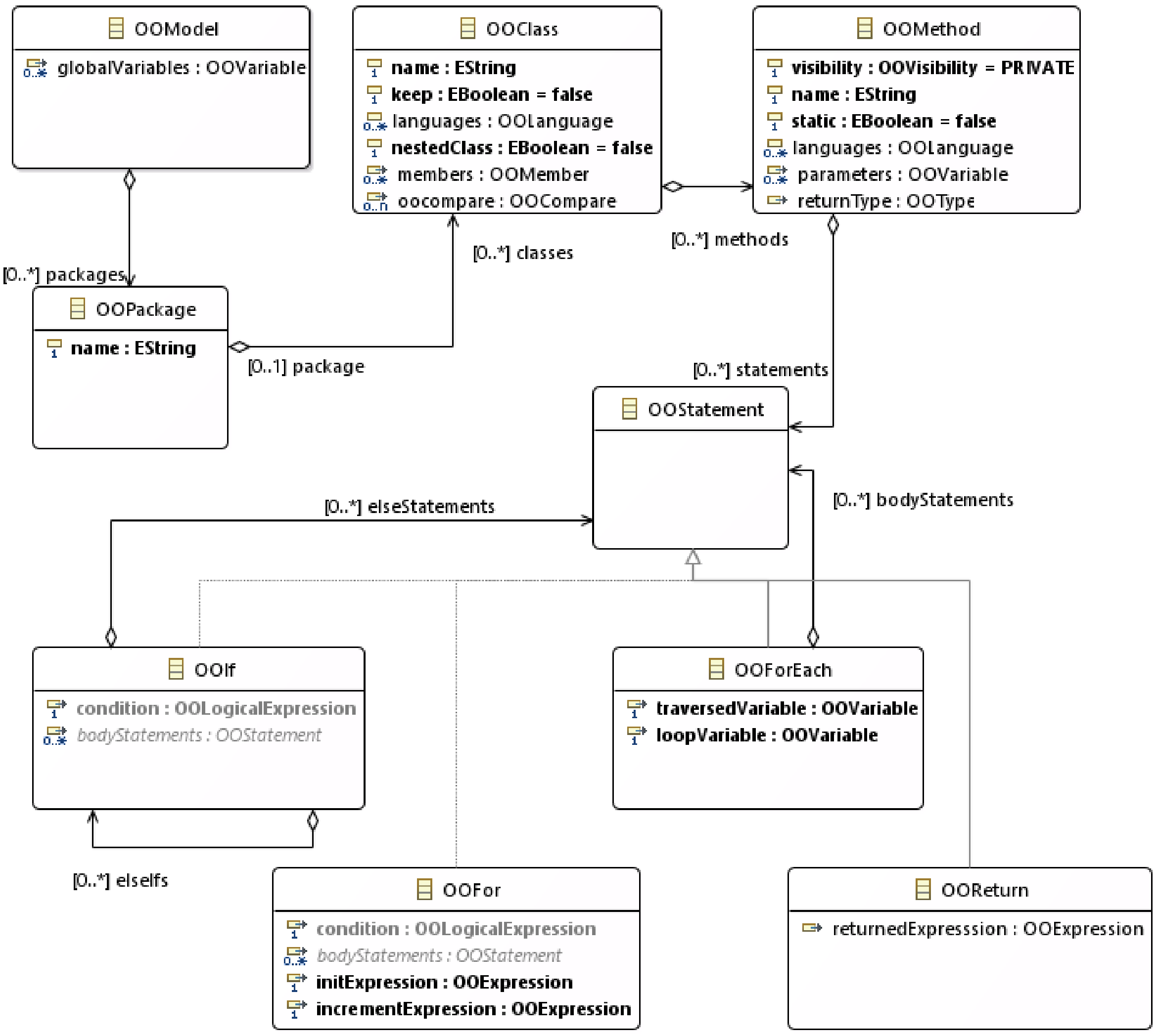
OOModel
globalVariables : OOVariable
OOClass
name : EString
keep : EBoolean = false
languages : OOLanguage
nestedClass : EBoolean = false
members : OOMember
oocompare : OOCompare
OOMethod
visibility : OOVisibility = PRIVATE
name : EString
static : EBoolean = false
languages : OOLanguage
parameters : OOVariable
returnType : OOType
[0..*] packages
[0..*] classes
[0..*] methods
OOPackage
name : EString
[0..1] package
[0..*] statements
OOStatement
[0..*] elseStatements
[0..*] bodyStatements
OOIf
condition : OOLogicalExpression
bodyStatements : OOStatement
OOForEach
traversedVariable : OOVariable
loopVariable : OOVariable
[0..*] elseIfs
OOFor
condition : OOLogicalExpression
bodyStatements : OOStatement
initExpression : OOExpression
incrementExpression : OOExpression
OOReturn
returnedExpresssion : OOExpression

Fig.2. A subset of the object-oriented code metamodel.

MDA is often criticized as being an idealistic forward engineering approach. Our method is also a forward engineering approach because changes in the code will not be reflected in the semantic model. Nevertheless, we believe that Domain-Specific Modeling complemented with manually written code is more efficient than general-purpose modeling. Therefore, our research and the method described herein, address Domain-Specific Modeling. In the common workflow of Domain-Specific Modeling, self-contained modules are generated that address specific domains of the whole system. These modules can then be easily integrated with manually written code. Because of this, generated code is practically never modified manually but by updating the DSM and regenerating the code.

The approach was successfully applied and greatly simplified multi-platform code generation in ProtoKit 2. However, it was not possible to represent the code generated for different target languages in a completely universal way. Section 5 explains the difficulties and the solutions we applied.

V. CHALLENGES

In this section, the challenges are explained that were faced during the application of our approach. We have identified three main difficulties that are described in the following subsections. For each challenge, we describe the nature of the problem and the possible solutions we considered.

A. Supporting Class Libraries

Class libraries are required for basic operations, such as reading a file or printing to the screen. Even simple programs require features from class libraries. However, supporting class libraries in our model-driven approach is challenging because of two reasons. The first reason is that the class libraries of different languages and platforms are different, so the universal nature of the model cannot be maintained. The second reason is that class library code is ordinary code like the program being modeled. They are instances of the same metamodel and class library model should be implicitly part of instance models despite that it is not a base of code generation. This relation is depicted in Figure 3.

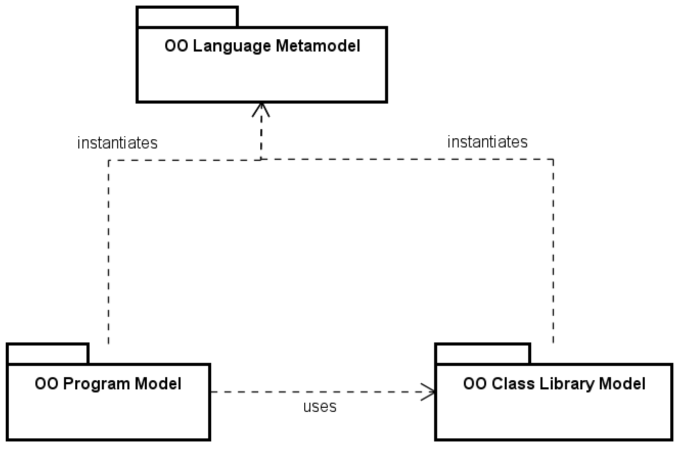

Fig.3. The relations of the program model and the class library.

Class libraries of widely used programming languages are extensive and complex and it is impractical to manually build a model about them. Nor is it considerably easier to build the model of a class library with a tool because it would require a complex parser to process the source code of the class library. Furthermore, not all class libraries are necessarily open sourced. We have considered the following solutions for this challenge:

(1) Introducing conditional elements and custom code snippets in the model: it is possible to add custom code snippets in the model to allow the inclusion of code into the output that would be impractical to model otherwise. If this is combined with the support of conditional elements, such as classes, methods or statements, that will make it possible to embed language-specific arbitrary pieces in the model. This approach is suboptimal since the model loses its universality but it is easy and fast to implement.

(2) Building model from class libraries: it requires complex tooling but it is possible to build a model from class libraries and reference classes from the library in the model of the program to generate. However, class libraries of different languages differ. We expect to solve this problem by introducing an additional transformation between the common model and the M2T transformation that generates code. This transformation step would add language-specific parts to the general model. Nevertheless, this mechanism requires further research.

(3) Integrating class library functionality in the language-independent metamodel: by analyzing several different programming languages, it is found that the dividing line between language and library is blurry. For example, in Basic, printing output to the screen is done by the PRINT statement, which is integral part of the language, whereas other languages tend to provide the same functionality in their standard libraries. Another example is how some compound data types are implemented. In PHP, arrays are maps that have integral keys and maps are part of the language not the class library. In turn, other languages usually support only conventional arrays with numeric indexes, and maps are supported by the standard library. Similarly, in the metamodel of object-oriented programs that we use to model the generated code, we are able to model some features as language elements despite that they are usually implemented in the class library. When the actual code generation happens in the M2T transformation, these can be mapped to class library calls. This approach is convenient for simple functionality, such as lists, sets, maps and other compound types or simple functionality from the class library, such as data conversion.

(4) Designing a universal abstract class library: we have also contemplated the possibility of designing a fictional class library for our object-oriented metamodel that could be used in models. This class library is unimplemented and does not have a concrete syntax, it only exists as a concept. By using the API of the class library in the models, we can model the behavior of programs. At a later stage, we define the mappings from this conceptual API to concrete APIs of target languages. This indirection allows us generating code to libraries of different platforms without requiring them to have a common interface. Like the second approach, this also requires an extra step in the transformation chain. This extra transformation and the need for mapping API calls to concrete libraries require more development effort in the tool but this mechanism can preserve the universal nature of the model.

In our prototype, we added some compound types, such as list, in the metamodel and in other cases, we added language-specific custom snippets. In the ProtoKit tool, these two were sufficient but we plan to lead further research on supporting class libraries either by parsing the class libraries of the languages or by creating an abstract class library.

B. Semantic Differences in Target Languages

Object-oriented GPLs share a large subset of common notions, such as class, static variable, instance method, constructor etc. Nevertheless, more thorough analysis reveals that there are features that only apply to a particular GPL or a set of GPLs. For example, C++ supports multiple inheritance and operator overloading, whereas Java supports neither. In turn, Java supports nested classes, default methods in interfaces and enums with methods. Another fundamental difference in the two languages is the semantics of how parameters are passed to methods. C++ determines the semantics by the parameter list in the definition of the method, whereas Java always passes variables of primitive type by value and objects by reference. We have considered the following solutions for this challenge:

(1) Introducing conditional elements in the model: it is possible to add qualifiers to the model and mark certain elements, such as methods, operators, statements language-specific. This helps dealing with some of the semantic differences, for example, we can define an operator and mark it as C++-specific. In turn, it is possible to add an equivalent method as Java-specific. Statements involving the operator or the method must also be added twice, marked as C++ and Java-specific, respectively. On the other hand, there are some semantic differences that cannot be efficiently solved with the mechanism of conditional model elements, such as default methods.

(2) Using a common subset of the features of target languages: the semantic differences could be avoided if we only used a common subset of features that have the same semantics in all of the target languages. For some scenarios, this approach indeed works. For example, in Java, we can perfectly live without nested classes. However, some of the language-specific features contribute to the usability and maintainability of the code. If the generated code does not leverage these features, for example operator overloading in C++ or Java enum methods, programmers will not feel comfortably when using the generated code. The generated code will not be perceived as well-designed code. Furthermore, some fundamental differences, such as the one regarding parameter passing cannot be avoided because methods and parameters are essential parts of any software program.

(3) Allow the union of all of the features of the target languages and define a mapping to supported features: the model of the program can support any features and this is later mapped to supported features. For example, in Java, nested classes can access members of the enclosing class. Nested classes can be mapped to independent classes in C++ and the privileged access can be ensured with friend methods. This mechanism works well, if a convenient mapping is found for all of the supported features. Nevertheless, some specific features are challenging. For example, passing an object by value is not possible in Java. If that is required, we must either pass a list of primitive variables from the object state or a clone of the original object.

In our reference implementation, we have used the first two mechanisms so far. Conditional elements are supported and some artifacts are only generated for only one of the target languages, whereas same statements have conditional equivalents for both. Furthermore, we limited the supported elements excluding the features that were not needed in our ProtoKit tool. We have not yet defined any language-specific mappings of features but we aim to do further research on this and improve our tool with this mechanism.

C. Conventional Differences in Target Languages

There are some differences between languages that are not strictly related to semantics but to conventions. For example, classes that define a natural order usually implement the Comparable interface and the compareTo() method in Java. It is technically possible to provide a compareTo() method in other languages, such as C++, the use of this method is a Java convention and is not natural in other languages. In C++, overloading the relational operators is the preferred way to define natural order. We have considered the following solutions for this challenge:

(1) Introducing conditional elements in the model: it is possible to add qualifiers to the model and mark certain elements, such as methods, operators, statements language-specific. This helps supporting conventions by marking the conventional additions as specific to a particular target language. This approach very well supports conventions. Nevertheless, it leads to duplication of elements that actually belong to the same function. For example, support of natural order would be implemented twice: first in the Java-specific compareTo() method, secondly in the C++-specific overridden operators.

(2) Supporting higher-level concepts in the metamodel: another idea for dealing with this difficulty is supporting higher-level concepts in the metamodel, such as semantical equality, natural order, cloning etc. These can determine how these features should work and these can then be mapped to concrete code in different target languages. The advantage of this approach is that it allows for handling these features at a single place. The disadvantage is that it requires an extra step in the transformation chain before the M2T transformation, where these concepts are mapped to concrete language constructs.

In our prototype, we have used conditional elements because they helped addressing several issues explained earlier. However, using these conditional elements deteriorate the readability of the model transformation and the model, so we plan to add support for higher-level concepts in future research.

VI. Conclusion

Despite these limitations, the approach still made our ProtoKit tool more flexible, better readable and less error-prone because of the following reasons:

(1) A large amount of code is still unconditional and thus generated universally from the same subset of the model.

(2) The templates are simpler and easier to read because of the general object-oriented model.

(3) Although there are conditional parts in the M2M transformation, their number is low and corresponding conditional parts are located near in the code. This makes it easy to understand the M2M transformation.

In the ProtoKit tool, our novel method has proven to be useful in achieving better flexibility, easier maintainability and more robust software that supports multi-platform code generation. Future research will explore more in depth the techniques that are described in Section 5 to aid the limitations. We believe that our method will contribute to the success of adopting model-driven approaches and generating code to multiple platforms. The results explained in the paper will be of great use for tool developers.

Acknowledgment

This work was partially supported by the CONCERTO (ART-2012-333053) EU-Artemis project, co-financed by the ARTEMIS Joint Undertaking and the Hungarian National Research, Development and Innovation Fund. This paper was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences and supported by the ÚNKP-16-4-III. New National Excellence Program of the Ministry of Human Capacities.

References

[1] D. S. Frankel, “Model Driven Architecture: Applying MDA to Enterprise Computing”, John Wiley & Sons, 2003.

[2] D. Thomas, “UML - Unified or Universal Modeling Language? UML2, OCL, MOF, EDOC - The Emperor Has Too Many Clothes”, Journal of Object Technology, vol. 2, no. 1, 2003, pp. 7-12.

[3] E. Seidewitz, “UML with meaning: executable modeling in foundational UML and the Alf action language”, Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, Portland, Oregon, USA, 2014, October 18-21.

[4] M. Fowler, “Domain-Specific Languages”, Addison-Wesley, 2010.

[5] S. Kelly and J. P. Tolvanen, “Domain-Specific Modeling: Enabling Full Code Generation”, Wiley-IEEE Computer Society Press, 2008.

[6] N. Harrand, F. Fleurey, B. Morin and K. E. Husa, “ThingML: a language and code generation framework for heterogeneous targets”, Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), Saint-malo, France, 2016 October 2-7.

[7] J. Stocq and J. Vanderdonckt, “A domain model-driven approach for producing user interfaces to multi-platform information systems”, Proceedings of the working conference on Advanced visual interfaces, Gallipoli, Italy, 2004 May 25-28.

[8] E. Umuhoza, H. Ed-douibi, M. Brambilla, J. Cabot and A. Bongio, “Automatic code generation for cross-platform, multi-device mobile apps: some reflections from an industrial experience”, Proceedings of the 3rd International Workshop on Mobile Development Lifecycle, Pittsburgh, PA, USA, 2015 October 26.

[9] E. Stroulia, B. Bazelli, J. W. Ng and T. Ng, “WL++: code generation of multi-platform mobile clients to RESTful back-ends”, Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems, Florence, Italy, 2015 May 16-17.

[10] G. Kövesdán, M. Asztalos and L. Lengyel, “Modeling Cloud Messaging with a Domain-Specific Modeling Language”, Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud, Valencia, Spain, 2014 September 30.

[11] The Object Management Group, “CORBA 3.3 Specification”, http://www.omg.org/spec/CORBA/3.3/.

[12] W. Grosso, “Java RMI”, O’Reilly Media, 2001.

[13] World Wide Web Consortium, “Simple Object Access Protocol (SOAP) Specification”, http://www.w3.org/TR/soap/.

[14] L. Richardson and S. Ruby, “RESTful Web Services”, O’Reilly Media, 2007.

[15] International Telecommunication Union, “ITU-T Z.100 Standard. Specification and Description Language (SDL)”, http://www.itu.int/ITU-T/studygroups/com10/languages/Z.100_1199.pdf.

[16] J. Quaresma, “A Protocol Implementation Generator, Master Thesis”, http://nordsecmob.aalto.fi/en/publications/theses_2010/jose_quaresma.pdf.

Authors’ Profiles

Gábor Kövesdán earned his MSc degree in computer engineering in 2013. Currently, he is a teaching assistant at the Department of Automation and Applied Informatics of the Budapest University of Technology and Economics and his area of interest is domain-specific modeling, model transformations and code generation.

László Lengyel received his PhD in 2006. He is an Associate Professor and fellow in the Department of Automation and Applied Informatics at the Budapest University of Technology and Economics. His various research fields focus on software development methods, the engineering of domain-specific languages, model-driven application development, cyber-physical systems, Internet of Things, and cloud-based solutions. The most important milestones in his professional career include, but are not limited to: the Bolyai János professorship (2007-2010 and 2015-2018), the Siemens Excellence Award (2008), and being chosen as the recipient of the NJSZT Kemény János-award (2012), BME Innovation Price (SensorHUB concept and framework) (2015), ÚNKP-16-4-III. New National Excellence Program of the Ministry of Human Capacities (2016-2017).

How to cite this paper: Gábor Kövesdán, László Lengyel, "Multi-Platform Code Generation Supported by Domain-Specific Modeling", International Journal of Information Technology and Computer Science(IJITCS), Vol.9, No.12, pp.11-18, 2017. DOI: 10.5815/ijitcs.2017.12.02