Guidelines for Improving Model To Text Transformations

Anderson Ledo[1], **Natã Melo**[1], **Franklin Ramalho**[1]

[1]Universidade Federal de Campina Grande (UFCG)
Rua Aprígio Veloso, 882, Bairro Universitário – Campina Grande – PB – Brazil

{ledo, natavm, franklin}@dsc.ufcg.edu.br

Abstract. *The application of good software engineering principles is a mandatory practice in order to achieve good quality artifacts. In this paper, we present a set of guidelines to be employed in the definition of MDA M2T transformations in order to archive good transformation definitions. We applied these guidelines to a real code generator in order to evaluate their impact and aplicability.*

1. Introduction

As the number of systems fully generated by the Model Driven Architecture (MDA) [Miller et al. 2003] infrastructure is continuously increasing, the usefulness of model driven methodologies becomes clearer and effective guidelines towards how to properly employ each piece of the MDA standard puzzle are required.

In an MDA transformation chain one frequently has to define (*i*) model-to-model transformations (M2M), which transform models from a certain domain into models of another one, and (*ii*) model-to-text transformations (M2T), which turn models into runnable assets. For instance, generating Java code from UML models would require a M2M transformation from UML to Java and then a M2T transformation from Java models to concrete code. Alternatively, one could directly transform UML models into Java concrete code following a template based approach. However, our focus is on fine grained transformations because they are more prone to be reused in different contexts, as advocated by Oldevik in [Oldevik et al. 2004].

The complexity involved in the task of elaborating a M2T transformation definition increases as much as the size of the metamodel comprising the elements to which the concrete syntax must be generated. Depending on the concepts that are metamodeled, this size can range from a few units to dozens or hundreds of elements. It is usual to see M2T transformations defined in some *ad hoc* way, coded with some type of *cowboy coding* style. This posture is motivated by the lack of knowledge about how to tackle the models complexity in M2T transformations. Not only novice transformation developers but also proficient ones can experience difficulties when faced to the task of elaborating transformation definitions for a great number of metaelements and their relationships.

The elaboration of methods and the sharing of experience among practitioners are key factors to the successful construction of a mature body of knowledge in the field of transformation definition. However, due to the evident novelty of the subject, defining good practices to be applied in transformation definitions are still a challenge.

and presents an approach for the elaboration of code generators focused on compiler techniques. Visser, in [Visser 2008], presents a DSL for implementing web applications and describes a process of model transformation and code generation based

on rewriting rules. Völter, in [Völter 2003] and [Völter 2004], presents guidelines and patterns which comprises recommendations for a better design of MDD artifacts, from metamodels to transformations. Czarnecki and Helsen, in [Czarnecki and Helsen 2006] present a survey of model transformation approaches, including M2T transformations, classified by its features. Some reusable patterns for M2M transformations have been proposed by Iacob [Iacob et al. 2008]. Oldevik, in [Oldevik et al. 2004], discusses the influence of the abstraction level of the models when defining M2T transformations. Although all these works deal with the issue of concrete code generation, they do not propose strategies, techniques or guidelines for the improvement of MDA M2T transformations in the code level.

Guidelines are directions to assist and to guide people in performing certain activities. In the computing context, we can mention guidelines for best-practices programming [DiMauro and Balena 2005] and for modeling a domain [Silingas 2006].In [Herczeg et al. 2009], guidelines prove to be effective as strategies to improve the performance of JavaScript programs even after the advent of just in time compilation engines.

In this work, we propose four guidelines for assisting transformers in the elaboration of M2T transformations. Our objective is that one can use them to produce clearer, more readable and understandable M2T transformations, consequently becoming easier to be maintained.

This paper is organized as follows. Section 2 gives an overview of MDA M2T transformations. Section 3 introduces the proposed guidelines, whereas Section 4 presents their application to the JavaCG tool [Ledo et al. 2008]. Finally, Section 5 concludes this paper with some considerations and remarks towards future works.

2. Metamodel Based Textual Transformations

Model to text (M2T) transformations support the process of concrete syntax generation from models. They are composed of a set of transformation definitions which are based on the concepts of a metamodel. A transformation definition has a collection of rules that describes how each metaelement from the metamodel must be transformed into one or more concrete syntax constructs of the target language. Examples of available M2T transformation languages are MOFScript [Oldevik 2006] and MOF2Text [MOF2Text 2008]. The Fig. 1 illustrates part of a model to text transformation, implemented in MOFScript, that generates Java concrete syntax. At line 2, there is the text transformation declaration named `JavaTT`, whose input metamodel is `"JavaAbstractSyntax"`, being referenced inside the text transformation as `jast`, at lines 4 and 8.

Generally, a transformation rule is defined for a specific metamodel element *i.e.*, the *context type* of the rule. However, it is also possible to define a transformation rule without a context type, *e.g.* for handling values passed as arguments. In addition, a return type can be specified for both kinds of rule. In Fig. 1, a rule is illustrated at lines 4-13. It extracts the properties from the `jast.ClassInstanceCreation` metaelement and combines its values with Java's syntactic constructs. At line 5, the reserved word `new` is outputted. This rule must be applied to instances of the `jast.ClassInstanceCreation` metaelement, which is its *context type*, as stated at line 4.

In a MOFScript rule, several common elements from traditional imperative lan-

guages are also present, like variables, print statements, iterative statements, conditional statements, etc. In Fig. 1, the variable `args` is declared for helping to concatenate the code's syntactic constructs to be generated, as illustrated at lines 7, 9, 10 and 12. At line 8, the iterator `forEach` is applied to the collection `self.arguments`. The persistence of the code is accomplished by the `print` statement, as at lines 5, 6 and 12.

```
1  import ''utilPkg/JavaTexttransformation.m2t''
2  texttransformation JavaTT (in jast:"JavaAbstractSyntax"){
3    ...
4    jast.ClassInstanceCreation::getInstanceConstruction():String{
5      print('' new '')
6      print(self.type.name.identifier)
7          var args : String = ''''
8          self.arguments->forEach(str : jast.StringLiteral){
9            args += ''\"'' + str.identifier + ''\"''
10           if(str != self.arguments.last()) args += '', ''
11         }
12         print(''('' + args + '')'')
13   } ... }
```

Figura 1. An example of MOFScript code.

3. Guidelines in Textual Transformations

The elaboration of the guidelines is mainly based on the development of the JavaCG project [Ledo et al. 2008], a code generator for the Java language, written in MOFScript. It takes, as input, models based on the JavaAbstractSyntax metamodel, from [AtlanModZoo 2010], and transforms them into concrete code. JavaCG was used in the automatic generation of JUnit tests [Gamma and Beck 2010] to check the conformance between UML class diagrams and their Java implementations [Pires et al. 2008]. More than 90 *KLoC* were generated using the JavaCG transformations in [Pires et al. 2008].

During the development of the JavaCG and other projects, we faced some difficulties in the evolution of the project as the number of metaelements being tackled increased. The strategies adopted to overcome the arosed problems are described in this section as a set of guidelines.

Separating the syntax definition from the handling of the model elements (G1)

Clearly different concerns of a code generator are: (1) the handling of the model elements and its related elements in order to generate their concrete syntax; and (2) the concrete syntax definition itself. Despite the clear distinction between these concerns they are frequently tightly coupled in the majority of the implementations. For instance, the transformation code illustrated in Fig. 1 shows scattered strings along the transformation rule. If the resulting concrete syntax of this rule needs to be changed then a transformer first needs to understand practically the whole algorithm logic of the rule and them locate the appropriate places that need to be updated in order to change the resulting concrete syntax. The handling of the models and the syntax definitions are excessively coupled.

Keeping model handling and syntax definition apart simplifies the code of the rules and makes them easier to understand and maintain. In addition, it becomes easier

to understand what the result of the rule will look like. In order to update the concrete syntax definition one should understand only the syntax of the target language.

```
1 jast.ClassInstanceCreation::getInstanceConstruction(){
2  var type : String = self.type.getType()
3  var args : String = getExpressionCollectionCode(self.arguments, ````,
       ``, '', ````)
4  result = classInstanceCreationTemplate(type, args)
5 }
```

Figura 2. Rule presented in Fig.1 after the application of the guidelines.

In Fig. 2 we illustrate the substitution of the scattered strings, in the version of the same rule in Fig. 1, by a calling to the rule `classInstanceCreationTemplate()` (line 4). The syntax definition for the *ClassInstanceCreation* metaelement is encapsulated inside the `classInstanceCreationTemplate()` rule, illustrated in Fig. 3. If one would like to change the syntax definition for a *ClassInstanceCreation* then he or she only needs to change the template rule. For instance, in order to change the constructs of a *classInstanceCreation*, one only needs to modify the the strings at line 2 of Fig. 3.

```
1 module::classInstanceCreationTemplate(type: String, args: String) :
      String {
2   result = `` new '' + type + ``('' + args + ``)''
3 }
```

Figura 3. Template rules for the definition of the syntax of a *ClassInstanceCreation* metaelement.

Tackling one metaelement per rule (G2)

The definition of a textual transformation for a given metaelement involves the definition of the transformations for its related elements. Frequently, the definition for these elements are tackled inside the same rule. If a second metaelement involves elements of the same type of the related elements of the metaelement, then the definitions for the same type of elements will have to be tackled again because the former transformation definitions were not modularized. Frequently, this problem is not detected in the beginning of the definition of the transformations, when just coding the definitions on demand seems more productive due to the simplicity of some elements.

Instead of tackling the transformation definition for several elements inside the same rule, one should define a separate rule for each element and call this rule whenever a transformation of an element of the same type is needed. Avoiding tackling the syntax generation of more than one element per transformation rule not only improves modularity but also simplifies the structure of the rules, making them easier to be understood.

For instance, in the rule `getInstanceConstruction()` defined at lines 4 - 13 of the Fig. 1 the concepts for a *Type* and for a *StringLiteral* metaelement, held by the variables `type` and `str`, are tackled at lines 6 and 9, respectively. At line 6, a definition for the *Type* metaelement is stated as the `identifier` attribute of the `name` reference of the `type` related element. At line 9, a definition for the *StringLiteral* metaelement

is stated as the value of its *identifier* surrounded with quotes. Both definitions for *Type* and *StringLiteral* should be abstracted by the calling to individual rules that define the transformation for these metaelements and could be reused in other places of the transformation. For instance, `getType()` and `getStringLiteralCode()`, respectively.

Another problem that can arise is that once an inline definition is used it may cover not all the possibilities of transformations. For instance, the inline definition for an instance of *Type* at line 6 of Fig. 1 is only suitable if `type` is a reference to a *SimpleType*, from the JavaAbstractSyntax metamodel. However, this is not the only possible reference type, *e.g.*, sometimes we can have an *ArrayType* element. Using the approach of not making inline definitions and then calling corresponding rules provides the advantage of using polymorphism for selecting the correct rules to be called according to the correct element instance types. Following this guideline allow us to cover this case only by providing new `getType()` rules for other metaelements (as context types). Examples of rules that provide this facility are presented in Fig. 4.

```
1  jast.SimpleType::getType(){
2    var name : String = self.name.getSimpleNameCode();
3    result = simpleTypeTemplate(name)
4  }
5  jast.ArrayType::getType(){
6    var componentType : String = self.componentType.getType()
7    var elementType : String = self.elementType.getType()
8    var dimensions : Integer = self.dimensions
9    result = arrayTypeTemplate( componentType, elementType, dimensions)
10 }
```

Figura 4. The 'getType()' rules

Defining separate rules for collections (G3)

Collections of elements have a particular syntax frequently defined by starting terminals, element separators, the syntactic representation for each element and the ending terminals. For instance, an array initializer in the Java language comprises a left curly brace "{" as a starting terminal, commas "," as value separators, the concrete syntax for each element of the array and a left curly brace "}" as an ending terminal.

For all the model elements, in a metamodel, with a multiplicity greater than one there is a collection to be tackled in a M2T transformation. Along with this collection come all the transformations for each single element and the combination of the results with the terminals. It is usual to write iterations inside the rule to get the code for each element and concatenate it with the terminals, but it is not the best approach, since more code should be written as more collection elements emerge.

Specific rules for collections of elements should be used instead of defining iterations for every element with multiplicity greater than one that has to be transformed. For instance, this situation is illustrated by the substitution of the iteration at lines 8-11 of Fig. 1 by the calling of the rule `getExpressionCollectionCode()`, defined in Fig. 5. This constitutes a more modular approach as repetitive iterations are avoided in the overall transformation. This rule can be reused for handling different collections of expressions (arithmetic, boolean, literal, etc.). It must be accomplished by (1) calling the

rule `getExpressionCode()` at line 4 of Fig. 5, which polymorphically selects the correct rule to be called according to the concrete type of the metaelement being handled, following **G3**, and (2) just changing the parameters that define the terminals and passing a list of expressions. An example of rule calling with different parameters can be seen at line 3 of Fig. 2, in which a collection of arguments is formed with starting and ending terminals as empty characters and with separator characters as commas.

```
1 module::getExpressionCollectionCode(exps: List, stt:String, sep:String,
      end:String) : String {
2   var code : String = stt
3   exps->forEach(e){
4     code += e.getExpressionCode()
5     if(e != exps.last()) code+= sep
6   }
7   result = code + end
8 }
```

Figura 5. Rule that generates the concrete syntax of a collection of expressions.

Avoiding the scattering of print statements (G4)

When print statements are scattered along the transformations the persistence of the code to the file is made by means of the collateral effects caused by them. The high number of collateral effects decreases the level of control over the transformation behavior. For instance, it is not possible to check any property of the generated code, *e.g.* if it has some specific syntactic construct, before persisting it, due to the fact that the collateral effects already persisted the code.

A good alternative to tackle this problem is to concatenate strings inside the transformation rules and to return them without printing it directly to a file. It allows the produced code to be checked before persisted, if needed. For instance, this strategy is used in the rules from Fig. 3, Fig. 2, Fig. 4 and Fig. 5, in which no strings are persisted but returned by the assignment to the `result` implicit variable.

4. Applying the Guidelines to JavaCG

In order to assess the effectiveness of the proposed guidelines we applied them to the JavaCG project. The application of the guidelines turned the transformations clearer in a sense that they became more easy to read, understand and maintain. As the readability, understandability and maintainability are software attributes not easy to be measured (frequently evaluated by experts qualitatively), we elected two metrics for measuring these attributes in a indicative manner, not conclusive though, and for summarizing the effect of the proposed guidelines over all the project:

Number of Statements per Rule (NSR) The number of statements counts the constructs and keywords. We advocate that a rule is easier to read and understand as less statements it has. For instance, the rule `getInstanceConstruction()` in Fig. 1 has 10 statements against 3 statements in the version of the same rule in Fig. 2. The readability may be subject to other factors too, but the number of statements showed to be simpler and suitable for the purpose of comparison in this paper.

Number of Meta-elements Tackled (NMT) We have advocated that handling many metaelements in a single transformation can preclude modularity and we have shown the influence of this concept with a rule example. In the `getInstanceConstruction()` rule we had three metaelements being tackled (*InstanceConstruction*, *Type*, *StringLiteral*). However, we should have only one (*InstanceConstruction*). A high value of this metric indicates a high level of redundancy since transformation definitions may be defined repetitively and it also indicates that the transformation is inconsistent since one can not guarantee the consistency among different definitions.

Before the application of the guidelines, the sum of NSRs, *i.e.*, the total number of statements for the whole transformations, was 259. After their application it decreased to 158. The smaller number of statements indicates the rules are simpler to be read and understood. The whole transformation became more readable. Before the application of the guidelines, the NMT was 63 and after the application it was 35, indicating a decreasing in the amount of redundant code and a better modularity since the transformation definitions for particular metaelements are better well defined and reused whenever they are necessary.

5. Conclusions

Based on our experience we have proposed a set of guidelines in order to help improving the quality of M2T transformations. We applied these guidelines to the JavaCG project and we assessed their effectiveness by discussing the improvements for a particular rule, and by measuring some metrics for the whole transformation. Despite the evaluation was focused on a Java code generator, the guidelines are not limited to the Java domain, they can be applied to M2T transformations independently of the domain described by the input metamodels.

To the best of our knowledge, there is no such kind of experience sharing about techniques or methods to improve MDA M2T transformations in the code level. Additionally, the proposed guidelines could be directly applied to M2T transformations written in the standard MOF2Text [MOF2Text 2008] language or they could be adapted to template-based technologies, such as Xpand [M2T-XPand 2008] and JET [JET 2007], in order to simplify templates definition. It is possible due the fact that (*i*) these technologies share common features to the MOFScript language, such as imperative semantics, rule abstractions, etc., and (*ii*) the proposed guidelines focus on simple features. For instance, one could create a number of solutions to simplify the definition of M2T transformations, *e.g.*, the use of an AOP approach, however, the use of simple language features makes the approach easier to be extended or adapted.

Future works include a better quantitative and qualitative evaluation of the effectiveness of theses guidelines and the proposal of possible new ones. They also may evolve to more specific and matured solutions and be described as patterns in order to be easier to learn and sistematically catalogued. In addition, we intend to turn available (*i*) the tool we used to perform the measurements; and (*ii*) a improved version to assist developers in the application of the proposed guidelines.

Acknowledgements

This work was supported by grants from CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico), FAPESQ/PB (Fundação de Apoio à Pesquisa do Estado da Paraíba) and PET (Programa de Educação Tutorial).

Referências

AtlanModZoo (2010). The AtlanMod Metamodel Zoo. *http://www.emn.fr/z-info/atlanmod/index.php/Zoos*.

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645.

DiMauro, G. and Balena, F. (2005). Practical Guidelines and Best Practices for Microsoft Visual Basic and Visual C# Developers (Pro-Developer (Paperback)).

Gamma, E. and Beck, K. (2010). JUnit. *At http://www. junit. org*.

Herczeg, Z., Lóki, G., Szirbucz, T., and Kiss, Á. (2009). Guidelines for JavaScript Programs: Are They Still Necessary?

Iacob, M., Steen, M., and Heerink, L. (2008). Reusable Model Transformation Patterns. In *Workshop on Models and Model-driven Methods for Enterprise Computing (3M4EC 2008))*.

JET (2007). Eclipse Modeling: Java Emitter Templates.

Ledo, A., Ramalho, F., Pires, W., and Melo, N. (2008). Javacg - a java code generator for the javaabstractsyntax metamodel. *http://code.google.com/p/java-cg/*.

M2T-XPand (2008). Xpand project.

Miller, J., Mukerji, J., et al. (2003). Mda guide version 1.0. *OMG Document: omg/2003-05-01, http://www. omg. org/mda/mda_files/MDA_Guide_Version1-0.pdf*.

MOF2Text (2008). Mof models to text transformation language. *http://www.omg.org/spec/MOFM2T/1.0/*.

Oldevik, J. (2006). MOFScript Eclipse Plug-In: Metamodel-Based Code Generation. In *Eclipse Technology Workshop (EtX) at ECOOP*, volume 2006.

Oldevik, J., Neple, T., and Aagedal, J. (2004). Model Abstraction versus Model to Text Transformation. *Computer Science at Kent*, page 188.

Pires, W., Brunet, J., Ramalho, F., and Serey, D. (2008). UML-based design test generation. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 735–740. ACM.

Silingas, D. (2006). Best practices for applying uml, part i. *http://www.magicdraw.com/files/whitepapers/*.

Visser, E. (2008). WebDSL: A case study in domain-specific language engineering. *Generative and Transformational Techniques in Software Engineering II*, pages 291–373.

Völter, M. (2003). A catalog of patterns for program generation. *Anais do, EuroPlop*.

Völter, M. (2004). Patterns for Model-Driven Software-Development.