MetaTT - A Metamodel Based Approach for Writing Textual Transformations

Anderson Ledo  
Software Practices Laboratory (SPLab)  
Federal University of Campina Grande  
Campina Grande, Brazil  
ledo@copin.ufcg.edu.br

Franklin Ramalho  
Software Practices Laboratory (SPLab)  
Federal University of Campina Grande  
Campina Grande, Brazil  
franklin@dsc.ufcg.edu.br

Natã Melo  
Software Practices Laboratory (SPLab)  
Federal University of Campina Grande  
Campina Grande, Brazil  
nata.melo@ccc.ufcg.edu.br

Abstract—We present MetaTT, a Metamodel based approach for writing Textual Transformations. It is an operational and stepwise approach for the construction of model-to-text generators in the context of Model-Driven Development. It relies on a particular architecture that fosters good transformation design and generates the main artifacts of a model-to-text generator, requiring the developer to take some design decisions and basically fulfill simplified templates. We evaluate its use by means of its application in the construction of a variety of instances of the problem, and we compared the generators built with and without MetaTT, taking into consideration the size and complexity of the metamodels being tackled.

I. INTRODUCTION

Model Driven Development (MDD) [1] is an established area in the software engineering academic and industrial communities. A set of different standards, tools and approaches has emerged through the years in order to realize the MDD proposal, such as Domain-Oriented Programming (DOP) [2], Microsoft’s Software Factories [3], Agile Model-Driven Development [4] and Model Driven Architecture (MDA) [6]. The envisioned high level of automation and reuse with the adoption of MDD can be fully reached if the models, metamodels and transformations are appropriately built and employed since producing and maintaining these artifacts demands a considerable team effort and development time. For instance, in order to construct a software for financial management it was necessary to conceive: (i) a metamodel to address the concepts concerning the financial management domain; (ii) another metamodel to address the concepts concerning the platform where the software will be deployed (such as a cloud computing environment, a mobile device or a desktop environment); and/or even (iii) another metamodel covering the concepts concerning the executable target programming language in which the software will be run. In addition, it would be necessary to elaborate the transformations in order to map the concepts of the different domains.

Some techniques and tools that focus on metamodeling and model transformation tasks have been proposed, such as [7], [8], [9] and [10], aiming at alleviating and guiding MDD activities in order to allow MDD applications to fully reach their potential of automation and reuse.

This work is focused on the task of writing model-to-text transformations (also called textual transformations or M2T transformations). In order to turn the models into text, a developer can use different approaches ranging from the combination of grammars with metamodels [11] and embedding of syntactic terminals into metamodels [12] to writing model-to-text (M2T) transformations. This latter is the more commonly adopted approach and has a wide variety of supporting technologies, such as MOF2Text [13], MOFScript [14], Acceleo [15], JET [16], XPand [17] and Epsilon [18].

Most of the known works address how the M2T transformations can be employed to generate concrete syntax in specific domains, but the problem is that none of them addresses how to systematically design and write good M2T transformations independently of the domain. Additionally, we have observed that this task becomes particularly harder when the metamodels on which they are applied are large and complex.

In this paper, we propose MetaTT, a Meta model based approach for writing Textual Transformations. It is an operational and stepwise technique for guiding the construction of the M2T transformations following the MDD concepts. MetaTT significantly decreases the effort employed by transformation designers in the construction of M2T transformations that deal with complex metamodels. It comprises an architecture addressing different concerns inside a M2T transformation engine. In addition, we provide a prototype tool support for automating the tasks prescribed by MetaTT.

This paper is organized as follows. Sec. II gives a brief overview about MDD, MOFScript and metamodels, pivotal elements for our approach. Sec. III describes the proposed architecture in order to build a M2T generator. Sec. IV presents the tool support provided to automate some MetaTT steps. Sec. V shows how we evaluated our approach by (i) comparing its usage against an (ad-hoc) approach. Sec. VI compares our approach to some related works. Finally, Sec. VII concludes this paper with some considerations and remarks towards future works.

II. BACKGROUND

In this section, we give a brief overview about MDD, MOFScript and Ecore, pivotal concepts and technologies in our approach.

We use MOFScript as our reference implementation language, but our approach can be used with other M2T transfor-
A. Model Driven Development (MDD)

MDD is a software engineering approach in which the key idea is to change the emphasis of the effort and time applied along the software development life cycle, focusing on activities such as modeling, metamodeling and model transformations, instead of focusing on the implementation as in the traditional practice.

The main elements of an MDD approach are: i) Metamodels, which describe how the models must be formed; ii) Models, which are instances of the metamodels; and iii) Transformations, which are rules that define how the input models must be transformed into the output models, according to their metamodels, or into text.

A simplified view of the MDD approach is shown in Fig. 1, an adaptation of the MDA framework presented in [5]. It shows a transformation chain from a high level input model (model 1) to the concrete syntax. Such model is in conformance with the metamodel 1 and it is taken by a transformation tool and turned into an output model (model 2), which is in conformance with the metamodel 2. A transformation tool executes a model-to-model transformation definition which transforms concepts from a language or domain (represented by the Metamodel 1) into concepts of another language or domain (represented by the Metamodel 2). Later, the output model is taken as an input model for a M2T transformation that maps it into a textual representation, i.e., the concrete syntax.

In spite some similarities between the two kind of transformations, the model-to-model transformations are essentially different from the M2T transformations. The former uses at least two metamodels (one for the input model and another one for the output model) and they are frequently written in hybrid languages, such as ATL [19]. On the other hand, M2T transformations commonly use only one metamodel and they are frequently written in imperative languages, such as MOFScript or MOF2Text.

B. MOFScript

MOFScript is a language for specifying M2T transformations that can be used to generate any kind of text, such as source code, documentation or markup languages. It is also semantically aligned with the OMG MOF Models To Text (MOF2Text) standard specification [13]. MOFScript’s features include mechanisms for abstraction and code reuse, such as inheritance, and OCL-like constructs.

In MOFScript, a transformation is specified by means of a transformation definition that comprises import declarations, a name, the definition of the input metamodel, some properties, variables and a set of rules. Fig. 2 illustrates a M2T transformation named JavaTT. At line 1, an auxiliary M2T transformation called JavaText transformation, which is located in the package utilPkg, is imported. This means that all its rules can be used inside the M2T transformation JavaTT. At line 3, there is the M2T transformation declaration, named as JavaTT whose input metamodel is "Java", being referenced inside the transformation as J. Lines 4-5 state the definition of a property and a variable, respectively. A property holds a constant value, while a variable can hold different values along transformations. MOFScript rules may have a context type, which is a model type to which the rule is applied. A rule also may have a return type, which can be one of the built-in MOFScript types (e.g., String, Real, etc.) or a model type. Parameters can be declared too. In Fig. 2, a rule is illustrated at lines 6-9. This rule returns a String value and it must be applied to instances of the JCompilationUnit, which is its context type.

Two other rules are shown in Fig. 2. The rule named toFile() at lines 10-13 is responsible for persisting strings into a file. In this rule there are the declarations of the parameters d (the directory to which the file should be saved), v (the value of the version of the file that will be saved) and code (the code to be stored), where v ranges values of Real type, while the remaining ones range values of String type. The rule named main at lines 14-17 is a special rule since it is always the first one to be executed in a M2T transformation and hence is called an entry point rule. Every M2T transformation just can be directly executed if it has a main rule. In Fig. 2 the main rule is executed for every CompilationUnit element occurring in the input model.

C. Metamodels

Metamodels are models that can be used to describe other models. In order to illustrate an application scenario of our approach, we have defined PLang, a simple metamodel that mirrors simple programming language concepts. It is shown in Fig. 3, where, in order to maintain simplicity, just a few composition relationships are shown. PLang metamodel works as follows:

- PLang models (ASTNode) are composed of a set of nodes (ASTNode). As in many procedural programming languages, PLang supports statements (Statement), expressions (Expression), declarations (Declaration), types (Type) and type declarations (TypedDeclaration).
- A statement may be further classified as an iterative statement (Iterative), an assignment (Assignment), a procedure call (ProcedureCall) or a conditional statement (ConditionalStatement).
- In PLang, one can declare functions (FunctionDeclaration), programs (ProgramDeclaration), variables (Vari-
import 'utilPkg/JavaTexttransformation.m2t'

texttransformation JavaTT(in J: 'Java')
   property dir : String = '/genCode'
   var version : Real = 1.0
   J.CompilationUnit::getCompilationUnitCode()
   : String
   var typeDeclarations : String = self.
       getTDCode()
   result = typeDeclarations
   }
module::toFile(d: String, v: Real, code : String)
   file(d+'JavaFile'+v++'.java')
   println(code)
   }
J.CompilationUnit::main()
   var code : String = self.
       getCompilationUnitCode()
   toFile(dir, version, code)
   }
...

Fig. 2. A MOFScript transformation example.

Fig. 3. PLang metamodel.

- VariableDeclaration and procedures (ProcedureDeclaration).
- TypeDeclarations may be classified as (1) a VariableDeclaration referring to the type of the values held by a variable; or as (2) a FunctionDeclaration referring to the type of the values returned as its result.
- In PLang, a ConditionalSttmnt is composed of (i) a condition that is an Expression; (ii) an ordered set of statements to be executed when the condition evaluates to false, the elseSttmnts.
  - A ProgramDeclarations is composed of other declarations plus a program name.

III. META TT

We propose MetaTT, a Metamodel based approach for writing M2T generators. It guides the organization, specification and control flow among M2T transformations from the information provided by a metamodel. MetaTT consists of an architecture that must be followed by the generated M2T transformations. Such architecture comprises the modules, sub-modules and contract that together aim at achieving a higher level of automation and reuse.

In Fig. 4 we illustrate how the developer can use MetaTT and how the resulting transformations can be used for the generation of concrete syntax from models. In region 1, we illustrate how the developer needs to interact with the MetaTT tool support (later detailed):

1) The developer provides a metamodel to the MetaTT tool.
2) MetaTT uses such metamodel for generating a set of M2T transformations that follow a standardized architecture prescribed by MetaTT.
3) The developer writes the rules (fulfilling rule stubs) previously generated in the Templates module.
4) The developer needs to inspect the transformations from the Main module and adjust them to reflect his or her design decisions in the M2T generator, such as the choice of the root element to be transformed.

In region 2 of Fig. 4, we illustrate how the generated M2T transformations are used to map models into concrete syntax. In this process, a model instance of the provided metamodel is given as input for the concrete syntax generator (the tool capable of performing the M2T transformations), generated by MetaTT, over the input model. As output of the concrete syntax generator, we have the generated files, such as documentation files, Java source code, XML files, etc.

A. Architecture

MetaTT organizes the M2T generator into three main modules: Main, Core and Templates, shown in Fig. 5. The Main module is responsible for starting the transformation and getting the resulting concrete syntax from Core module as well as persisting it. The Core module extracts information from the input models and uses the syntax definition from Templates module that in turn is responsible for defining the concrete syntax.

As can be noted in Fig. 5, the Core and Templates modules are further composed of other submodules. The Templates module does not depend on any other module, whereas the Main module plays the client role of the Core module, which concentrates the text generation process. Each module and corresponding submodules are explained in the next subsections.
1) **Templates**: This module provides the concrete syntax definition for a target language. For instance, considering Java programming language as the target language, this module will contain the specification of the concrete syntax to the method signatures, the type declarations, the keywords, etc.

This module comprises a set of *template rules* and *symbol tables*. The former provides the syntax definition for the non-terminal elements appearing in the metamodel to which the concrete syntax must be generated. The latter provides individual pieces of syntax for terminal elements that are not metamodeled as metaelements, e.g., keywords, separator characters, block delimiters, etc.

Whenever concrete syntax information is required in the text generation, this information is obtained from the Templates module by means of the `ITextualDefinitions` and `ITextualDelimiters` interfaces that give access to the `TextualDefinitions` and to the `TerminalSymbols` sub-modules, respectively. The `TextualDefinitions` sub-module contains the *template rules*, where each one of them defines a particular template for each metaclass of the metamodel. The `SymbolTerminals` sub-module contains the *symbol tables* that contain the string constants corresponding to language terminals (e.g., braces, separators, keywords, etc.). Whenever the rules in the `TextualDefinitions` sub-module needs terminal symbols, it accesses the `SymbolTerminals` sub-module through the `ITerminals` interface.

In Fig. 6 we illustrate part of the symbol table, in which each symbol is defined as a string constant that holds characters to be used in the syntax definition. At lines 1-4, commonly used characters are illustrated (left and right parentheses, left and right curly braces). Such characters can be reused in many template rules and other definitions (e.g., at lines 9 and 11...
and, later, in Fig. 8). At lines 6 and 7, specific keywords of the PLang \textit{ConditionalStmt} element are shown. At lines 9-11, some characters used in block of declarations, such as left curly brace (line 9), end line character followed by a tab character (line 10) or right curly brace (line 11), are shown.

When a M2T transformation developer needs to specify the syntax definition for a given element in MetaTT, he or she needs to define the values of the terminal elements in the \textit{symbol tables} and to define how the \textit{template rules} should arrange these elements in order to form the syntax for the non-terminal ones. For instance, suppose the developer has decided that a PLang \textit{ConditionalStmt} should have a syntax definition according the the excerpt of a BNF grammar shown in Fig. 7 (in which non-terminal elements are in bold font and terminal elements are in normal font), then he or she defines its concrete syntax as a \textit{template rule} as shown in Fig. 8.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig7.png}
\caption{An excerpt of a BNF grammar for the PLang \textit{ConditionalStmt}.}
\end{figure}

\begin{figure}[h]
\centering
\begin{verbatim}
module:: conditionalStmtTemplate(condition : String, thenStmts : String, elseStmts : String) : String{
    var code : String = 'if
    code += IF+LEFT_PARENTHESIS+condition+
    code += RIGHT_PARENTHESIS
    code += LEFT CURLY_BRACES
    code += thenStmts+
    code += RIGHT CURLY_BRACES

    if( not elseStmts.trim() = ' ' ' ){
        code += ELSE+
        code += elseStmts
    }
    code += RIGHT CURLY_BRACES
    } result = code

\end{verbatim}
\caption{Implementation of a template rule for the BNF grammar described in Fig. 7.}
\end{figure}

Fig. 8 illustrates the \textit{conditionalStmtTemplate} rule, that receives the previously extracted concrete syntax of three related elements: (i) a condition, (ii) the then block statement and (iii) the else block statement. At lines 4-7 these elements are combined with terminal strings (defined in the \textit{Terminal-Symbols module}) in order to form the concrete syntax for the conditional statement. At lines 9-13, the \textit{else} statement is appended if it is not an empty string. An example of the text output of the \textit{template rule} in Fig. 8 is the code in Fig. 9.

When some change in the concrete syntax specification needs to be done, it is accomplished by the modification of a well defined and/or localized rule and a symbol table. As a result, one can update the syntax to be generated by only updating the artifacts of the \textit{Templates module} or just exchanging this module by another one (respecting the interface contracts), without having to change any other modules.

Although the given example is related to PLang, that represents programming language concepts, our approach is not restricted to generate text only to programming languages.

It is also important to highlight that although we use a BNF grammar as a description of the syntax in our example, we do not enforce any formation rule in our templates so that the developer is free to change it as he or she desires. For instance, if the developer needs a M2T generator for HTML, he or she needs (1) to provide a HTML metamodel to MetaTT, (2) to adjust the Main module to transform the root element of the HTML metamodel (that may be the element named “html”) and (3) to fulfill the template rules and to define the terminals. In order to do activity (3) he or she does not depend on a grammar description. In fact, the description of the syntax is defined by the developer.\textit{HTML} syntax.

2) \textit{Core}: This module provides rules for carrying out the textual transformation of each metaclass of the metamodel. It is further divided into two sub-modules: \textit{Extractor} and \textit{Collections}.

\textit{Extractor} sub-module is responsible for the extraction of the concrete syntax information of each metaclass of the metamodel, whereas \textit{Collections} sub-module manages the extraction of the concrete syntax information for collections of instances of a given metaclass. Both sub-modules depend on each other since they are responsible for complementary functionalities. Whenever textual information for a collection of elements is required inside the \textit{Extractor} sub-module, it accesses the functionality provided by the \textit{Collections} sub-module through the \textit{CollectionExtractor} interface and whenever the transformation of a single element is required inside the \textit{Collection} sub-module it accesses the functionality of the \textit{Extractor} sub-module through the \textit{TextExtractor} interface.

The rules in \textit{Extractor} sub-module are named \textit{extractor rules}. For each metaclass of the metamodel there is an extractor rule responsible for invoking rules capable of extracting the concrete syntax pieces (one for each attribute or reference of that metaclass) that together will form the whole syntax to that metaclass. The task of combining these syntax pieces is the role of the template rules (through the \textit{TextualDefinitions interface}) from the \textit{Templates} module. For instance, for the \textit{ConditionalStmt} metaclass, illustrated in

\begin{verbatim}
if (a < b){
    println(‘’a is smaller than b’’)
} else{
    println(‘’a is greater than or equal to b’’)
}
\end{verbatim}

Fig. 9. Example of a code output from the \textit{template rule} in Fig. 8.
Fig. 10, there is an extractor rule, presented in Fig. 11. One can perceive the close relationship between the composition relationships of the ConditionalStmtnt and the structure of the code presented in Fig. 11. For each relationship from the ConditionalStmtnt metaclass targeting a Statement or an Expression, there is a rule invocation in the extractor rule, i.e., there is a correspondence between the member ends condition, thenStmtnts and elseStmtnts from Fig. 10 and lines 2, 3 and 4, respectively, from Fig. 11. At line 5, a template rule, named conditionalStatementTemplate, is invoked to combine the syntax pieces captured at lines 2-4 forming the whole concrete syntax of the ConditionalStmtnt.

At lines 3 and 4, in Fig. 11, some collection rules are invoked. Such rules are provided by the Collections submodule. They are responsible for abstracting the processing of collections of elements and enclosing them with their corresponding textual delimiters, obtained from the Templates module through the required interface TextualDelimiters. In this example, the collections are required since a collection of then and else statements may be involved in a conditional statement. By separating these kinds of rules into a different module, MetaTT helps simplifying the extractor rules.

3) Main: This module comprises the start and the finish point of the M2T transformation. It contains an entry point rule that receives an input model, identifies the metaclass this model is rooted and invokes the corresponding rule (an extractor rule already specified in the Core module) for that metaclass. Fig. 12 and Fig. 13 are two slightly different examples of main rules. In both, the declaration of the context type of the rule (at line 1) allows an element to be automatically selected from the input model. At line 1, in Fig. 12, the transformation matches with ProgramDeclaration elements and invokes the extractor rule named getProgramDeclarationCode() (at line 2) from the Core module. Persistence is made at lines 3-4. Fig. 13 is analogous to Fig. 12, but it performs the transformation for FunctionDeclaration elements.

A better control of the generation process is possible because the persistence is isolated in one module (Main), whereas the extraction of the text is isolated in another one (Core). The Main module is the only place in the whole text
Regarding the methods used to elaborate the M2T generator, one could use MetaTT or an ad-hoc one. We observed that, when no previous method for designing and coding the transformations is specified, a common behavior arises: The developer has some model to be transformed, he or she starts writing transformation rules for the root metaelements and iteratively adds more transformation rules for the other metaelements on demand, this is what we define as an ad-hoc approach. It is important to highlight that since we have not found another method for designing M2T transformations, the preliminary evaluation was focused on comparing MetaTT with the aforementioned ad-hoc method.

The subjects that performed the evaluation were two students from SPLab/UFCG: a graduate one who performed scenario 1 and 2 and an undergraduate one who performed scenarios 3 and 4, both with a background in MDD/MDA and with previous experience with M2T transformations and MOFScript. They received instructions about the MetaTT approach before using it.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Method</th>
<th>Metamodel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ad-hoc</td>
<td>Java</td>
</tr>
<tr>
<td>2</td>
<td>MetaTT</td>
<td>Java</td>
</tr>
<tr>
<td>3</td>
<td>Ad-hoc</td>
<td>PetriNet</td>
</tr>
<tr>
<td>4</td>
<td>MetaTT</td>
<td>PetriNet</td>
</tr>
</tbody>
</table>

TABLE I
THE POSSIBLE SCENARIOS THAT COMBINE THE TWO METAMODELS AND THE USE (OR NOT) OF META TT.

Tab. II shows three metrics about the inner structure of the metamodels. Together, these metrics give us insights about the effort necessary to write a M2T generator for each metamodel. The metrics and what they represent in the construction of the M2T generators are described as follows.

**Number of Metaclasses (NM).** Denotes the size of the metamodel and may be used to compare the amount of required effort to build M2T generators for a metamodel.

**Mean Number of References per Metaclass (MNRM).** Indicates the level of dependency between metaclasses. For instance, if a metaclass has three references to other metaclass(es), such dependency must be reflected on the structure of the M2T generator because the transformation of the former metaclass depends on the transformation of the latter ones. Therefore, the MNRM captures the mean number of conceptual dependencies that are to be tackled in the generator.

**Mean of the Depths of Inheritance Tree (MDIT).** The depth of inheritance tree (DIT) metric provides a measure of the inheritance levels from the top class in the hierarchy to a particular class [24]. For instance, in a design of classes which have a top class C, all the classes that directly inherit from C have a DIT of 1; if an additional class inherits a class that is child of C, this new class will have a DIT of 2. In PLang, the ConditionalSttmnt element (see Fig. 3) has a DIT of 2 because it inherits from Statement and hence from ASTNode. In order to represent this information for the whole metamodel we use MDIT, the mean of the DITs of the metaclasses, that
can be measured by the summation of the DIT measure for each metaclass in the metamodel divided by the number of metaclasses. In the construction of the M2T generators, the impact of the DITs is directly related to the need for querying the correct type of the instances to be transformed along the process of M2T transformation in order to choose the correct rule to transform that element. For example, an instance of the metamodel may have a set of abstract elements whose real type may only be determined in runtime. Therefore, the DIT measure for a given metaclass denotes the amount of effort necessary for the M2T generator to discover its correct type and to invoke the correct transformation rule.

NM is used to denote the size of the metamodels and MNRM and MDIT are used to denote their complexity. However, the complexity metrics should not be analyzed apart from the size metric. For instance, the MNRM values for both metamodels (Java and Petri Net) seem to be very close if analyzed without taking NM into account. Here, the number of metaclasses amplifies the impact of the measured attributes into the M2T transformations.

<table>
<thead>
<tr>
<th>Metamodel</th>
<th>NM</th>
<th>MNRM</th>
<th>MDIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Petri Net</td>
<td>99</td>
<td>1.57</td>
<td>0.71</td>
</tr>
<tr>
<td>Java</td>
<td>1.62</td>
<td>1.82</td>
<td></td>
</tr>
</tbody>
</table>

TABLE II SUMMARY OF THE METRICS FOR THE META_MODELS.

By applying the aforementioned metrics on two well-known and consolidated metamodels (Petri Nets and Java), we could observe four different scenarios:

**Scenario 1. Building a Java M2T generator following an ad-hoc approach.** Since the Java metamodel is big and complex (as indicated in Tab. II), the building of the M2T generator was intrinsically hard because the developer had firstly to understand the details of the metamodel. We observed that the code of the transformations became hard to understand and maintain. The reasons for such difficulty are tightly associated to the high rate of dependency among metaclasses along with the depth of the inheritance trees. The dependency among metaclasses led to many occurrences of nested transformation inside the rules, i.e., the concepts of the related elements were being tackled inside the same rule, leading to code repetition (because the same concepts are also required in other transformation rules) and inconsistency (because there was no guarantee that the same kind of metaelements are tackled equally in the repeated code). Avoiding code repetition by means of defining a proper rule for each metaclass and invoking it was hard at this stage because the conceptual dependency among metaclasses is too complex (e.g., the related elements also have other related elements, that in turn have more related elements and so on).

**Scenario 2. Building a Java M2T generator following the MetaTT approach.** For building the M2T generator, the developer first understood the concepts of the MetaTT approach. Then, the tool support could be used. Since the developer understood the structure of the rules and iteratively filled the templates in the Textual Definitions sub-module, we can consider the building process easier. The understanding of the M2T generator is tied to the understanding of the MetaTT approach. Maintaining the M2T generator is easier than by adopting the ad-hoc approach once it follows a modular design and it is automatically documented through our approach. The trade-off between learning the MetaTT concepts and the consequent improvement in the understanding and maintenance of the code generator was positive. The use of MetaTT also led to a significant improvement on the quality of the code of the rules, as reported in an preliminary study in [25], where we measured that (i) the number of metaclasses tackled per rule decreased 44%, which means that we decreased the amount of code repetition and improved the modularity of the rules and (ii) the number of statements per rule decreased 39%, which means that the code of the transformations became simpler to read and understand. Additionally, the produced Java code generator is used as part of two MDA-based tools in testing [26] and design conformance [27] projects.

**Scenario 3. Building a Petri Net M2T generator following an ad-hoc approach.** Since the metamodel is small (7 metaclasses) along with simpler concepts (a lower number of references per metaclass, 1.57, indicating less concept dependency), it was easy to start building of the M2T generator. Iteratively coding the transformation rules worked well in this case. The low complexity of the problem made a modular and well structured design an optional feature. The maintainability of the few transformation rules was not seriously impaired too. Understanding the code was also simple. This implementation has been used in the MDAVeritas project [28].

**Scenario 4. Building a Petri Net M2T generator following the MetaTT approach.** For building the code generator, there was the need to firstly understand the concepts of the approach and to understand the concepts of the metamodel. Due to the simplicity of the metamodel to be transformed, the overhead of adding the understanding of the approach was negative because it did not bring enough benefits. The understanding of the code generator is subject to the understanding of the aspects of the approach (structure of the rules, architecture and execution flow among them). Although the approach is documented, the effort necessary to understand and effectively employ it did not seem to be worthwhile when the metamodel is too simple. Maintenance became a simple task with the approach but not enough to justify its employment. This version of the M2T generator is available at [28].

The analysis of the described scenarios reveals that MetaTT is useful for tackling the complexity of writing M2T generators for big and complex metamodels (high number of metaclasses and relationships). However, when the metamodels are very simple (low number of metaclasses and relationships) it adds a possible unnecessary complexity to the task. Although writing ad-hoc M2T generators for small metamodels may seem reasonable, it does not seem to scale if the metamodel evolves and new concepts are added to it.

Further studies are necessary in order to define some
thresholds relative to the characteristics of the metamodel and the efficiency of our approach. For instance, we still cannot determine the point into which using MetaTT starts being more efficient.

**Threats to Validity.** The fact that only a few metamodels have been considered is an external threat validity as well as the background and experience (on M2T transformations) of the participants. In addition, we also identified the following threats:

- The metrics used to denote size and complexity are threatened by the lack of a rigorous metric validation. However, this threat is softened by: (i) the fact that they are already used to assess UML class diagrams [27]; and (ii) the correspondence between the concepts (dependency, composition, inheritance, etc.) in both domains ranges from very close to equal.
- The representativeness of the analyzed metamodels as “small” and “big” instances. In spite of the fact that they are comparatively different in size, we cannot strongly state that the PetriNet and the Java metamodels are representative instances of a small and big metamodel, respectively. Such notion would require a deeper analysis about the characteristics of a bigger set of metamodels.
- Although the subjects are not individuals randomly selected from the model driven communities, they are students enrolled in the model driven research field.

VI. RELATED WORK

In order to provide concrete syntax in an MDD approach, several works have been proposed.

[12] presents a technique for specifying concrete syntax by means of metamodels that are used as a basis in the process of text generation and model instantiation, allowing bidirectional syntax mappings (abstract syntax ↔ concrete syntax). The main difference from our approach is that they make mappings through the embedding of concrete syntax into metamodels, while our approach is not intrusive to the metamodels. The main advantage of having a template based approach is that one can have more control over the text generation process and more easily maintain the templates, since no formal specification is needed.

TCS [29], XText [30] and EMFText [31] are tools that support the definition of DSLs. Some of them include the generation of plugins, parsers, code editors, etc. These tools also support declarative mappings between abstract models and concrete syntax by means of grammar-like specification languages. Such mappings enable the automatic instantiation of models from the concrete syntax as well as the automatic generation of concrete syntax from the models. However, whenever a developer needs a finer control over the process of concrete syntax generation, he or she must adopt imperative M2T languages, such as MOFScript, Acceleo, XPand and JET, which have a set of mechanisms specially dedicated to concrete syntax generation. MetaTT does not propose a new tool for the specification of DSLs syntax or a new M2T language, but it proposes an architecture for designing and implementing M2T transformations towards a easier maintenance and evolution.

[11] proposes a technique to generate some IDE artifacts (like parsers and grammars) from an abstract syntax model (ASM or a metamodel). This technique leverages the process of IDE DSL development. For accomplishing such leveraging, the authors advocate the use of metamodels whose instances follow a tree structure. Our work is not focused on the complete generation of IDE artifacts, it is focused on the M2T transformations instead. Additionally, we do not impose restrictions on the structure of the metamodel to which we are generating text.

In all approaches above, a concrete syntax specification is required, being a grammar, a metamodel, or a DSL. This facilitates not only the process of text generation, but also the process of model instantiation from text. However, their applicability is limited to the scenarios described individually in each work.

[32] presents an approach for generating Java code from Fujaba models. Similarly to our work, it takes a template based generation approach and does not assume a concrete syntax model. It was not proposed to be a generic approach for a variety of metamodels and target syntaxes as our work does.

[33] presents a case study of code generation by model transformation. It shows how modularization principles and extensibility have been applied in the conception of WebDSL, a DSL for the elaboration of dynamic web applications with rich data. They concern all the pipeline of transformations (from the higher level to the lower one). While they propose a new approach addressing each part of a complete transformation chain, we focus on MDD M2T transformations and propose an approach to be integrated into existing model driven transformation chains, taking advantage of generation templates and clearly separating generated code and models.

The works aforementioned concern the generation of concrete syntax, but, most of them (1) are not focused on the systematic specification of M2T transformations aligned to MDD compliant technologies or (2) are not designed to be domain independent. In addition, our approach is the only one that is based on a proposed architecture with well-defined modules, responsibilities and contracts. This architecture aims at reaching a greater level of reuse and maintenance in the M2T transformations comprised by each one of its modules.

VII. CONCLUSIONS

MetaTT is a stepwise approach for building M2T generators. It is not domain specific, i.e., it may be adopted to build the M2T transformations able to generate the concrete syntax for MDD applications independently of the input and output languages involved in the transformation chain.

To the best of our knowledge, there is no similar experience about how to address M2T transformations independently of the domain and platforms involved in the transformation chain.

It is important to emphasize that MetaTT is able to generate text for models described in any level of abstraction. It is
just required that these models are in conformance with the metamodel from which the text must be generated. Regarding the evolution of the M2T transformations, since they are highly coupled with the metamodels, whenever a change is performed into the latter the former must have to be evolved too. As the rules defined in MetaTT are very close to the metamodel definition, changes in the latter may be reflected in the code of the M2T transformations through the re-generation of the transformations.

We do not propose ways to preserve the changes made in the code of the M2T transformations after the generation of the first version of the M2T generator by MetaTT. The generated text would be protected since the implementation language of the M2T transformations already provides mechanisms for such, but this is not our focus.

The MetaTT architecture was conceived in order to realize good transformations design and to enable the rules to be simpler and readable. During its elaboration, we performed constant improvements and selected good solutions that we have experimented. Part of the incorporated design decisions was inspired by the study provided in [25], where some guidelines about how to write M2T transformations are introduced.

The evolution of the transformations continues to be a responsibility of the developer, although it is softened by good design principles incorporated into MetaTT (e.g., modularity and separation of concerns). For instance, if a meta-element changes, the developer must reflect the changes in the M2T transformations. However, this activity is softened because the rules to be tackled in the transformation of each meta-element are well defined and separated through the architecture.

For the lack of space, we do not detail in this paper how the artifacts in the Core module are generated. However, it is important to highlight that our approach includes the automatic generation of the rules inside this module (as seen in Sec. IV) what decreases the effort employed by the developer for building M2T generators.

As future work, we intend to apply a deeper evaluation with a more controlled environment to MetaTT. This study involves also to investigate and propose new metrics to analyze the possible benefits and drawbacks in applying our solution, in particular those concerning maintainability.

Acknowledgments. This work was supported by The Conselho Nacional de Pesquisa e Desenvolvimento (CNPq) and by the Fundação de Apoio à Pesquisa do Estado da Paraíba (FAPESQ/PB). We also thank to PET/UFCG for the collaboration in this work.

References