Toward Standardised Model to Text Transformations

Jon Oldevik, Tor Neple, Roy Grønmo, Jan Aagedal, and Arne-J. Berre

SINTEF Information and Communication Technology,
Forskningsveien 1, 0373 OSLO, Norway
{jon.oldevik, tor.neple, roy.gronmo, jan.aagedal,
arne.berre}@sintef.no
http://www.sintef.no

Abstract. The objective of this work is to assess the qualities of the MOFScript language, which has recently been submitted to the OMG as a proposed model to text transformation language. This is done by identifying requirements for this type of language and evaluating the MOFScript language with regard to these. The language is presented along with a tool implementation and compared with the alternative languages submitted to the OMG Model to Text RFP.

1 Introduction

Ever since dawn of software modelling, technologies have been around to provide mappings from software models to useful technology platforms, such as databases, implementation languages etc. Along with the maturity of the modelling domain, the standardisation of modelling languages and technologies such as UML and the Meta Object Facility (MOF)[1], and the adoption of these technologies in practical use, the need for standardising transformation and mappings of these models has become apparent.

This need is currently being addressed through the ongoing standardisation activities in OMG concerning model to model transformations (MOF Query, View and Transformations – QVT)[2] and model to text transformations[4]).

The MOFScript language has been submitted as a proposal for a model to text transformation language to the OMG. This paper identifies different requirements for model to text transformation languages and evaluates the MOFScript language and tool against those requirements. There are three competing languages to MOFScript which are also discussed with regard to the requirements. The rest of this paper is structured as follows: Chapter 0 gives some background in the area of model to text transformation. Chapter 0 describes a set of requirements for model to text transformation languages. Chapter 0 describes the details of the MOFScript language and tool and gives a brief evaluation. Chapter 0 describes related work and chapter 0 concludes.

2 Background

Traditionally models have been used in software development to define and understand the problem domain or the different aspects of a system's architecture. After the

A. Hartman and D. Kreische (Eds.): ECMDA-FA 2005, LNCS 3748, pp. 239 – 253, 2005.
© Springer-Verlag Berlin Heidelberg 2005

modelling, one dove into implementation of the system without updating the models based on the actual implementation that was made. This issue is remedied within the MDA paradigm where the models are the prime artefact. From these artefacts large portions of the source code for the system can be generated.

The issue of generating code from models can be abstracted to the term *model to text* transformations as opposed to *model to model* transformations. The goal is to be able to create textual artefacts based on model information. Textual artefacts include other things than source code, such as various types of documentation.

Typically a code generator will not be able to generate all of the code that is needed to implement a system. Certain facets of a system, e.g. the static parts, are well suited for code generation, while there are challenges in modelling the more dynamic parts, for instance method bodies of classes. This means that parts of the source code for a system will be generated from the models while other parts will be hand crafted. The ability to protect the hand crafted parts of the code from subsequent code generation passes is important. In some cases one even may want to update the model based on changes made in the source code. One may say that the same issue is relevant for document generation, where one typically may want to add writings that are not part of the model to the resulting document.

The task of writing model to text transformation definitions will probably not be carried out by all software developers. However, it is important that the language used for such definitions is as easy to use as possible, e.g. by sharing properties with common programming or scripting languages. The usability of the language can made better through good tool support including features such as code completion, a feature present in most integrated development environments (IDEs) for normal programming languages today.

3 Requirements for Model to Text Transformation Languages

In the Request for Proposal (RFP) for MOF Model to Text Transformation[4], a set of requirements to such languages is identified. These are high-level requirements that provide a framework for defining a language that will fit the OMG way of thinking and align well with already adopted OMG specifications. The essential requirements that must be met include the basic ability of generating text from models, specifying transformations based on metamodels, the ability to specify complex transformations, the ability to allow multiple input models/metamodels for transformations, support for text manipulation functions, and reuse of existing standards, such as QVT, Object Constraint Language (OCL)[12], and MOF.

In addition, there are other obvious requirements, such as the ability to generate text to files, and the ability to query model element properties, which need to be supported by a model to text transformation language.

We acknowledge the OMG requirements as essential basic properties, and extend with a set of additional requirements that we deem desirable for a model to text transformation language. Some of these were previously identified in [6].

1. *Structuring:* The language should support structuring and control of text generation. This means that it should be possible to specify structures that orchestrate a set of finer grained text generations.
2. *Control mechanisms*: It should provide basic control flow mechanisms. This implies that it must be possible to provide the semantic equivalent of loops and conditional statements.
3. *Mix code and clear text*: It should provide a simple way of combining transformation code (logic), model data, and clear text. It shall also provide a way of converting model data to strings and use this in produced text
4. *System services*: It should provide support for string manipulation functions. It should also provide the ability to interact with system services or library functions, e.g. inquiring about the current date and time.
5. *Ease-of-use*: The concrete language should show similarity with existing well known approaches in order to be easy to use (such as programming or scripting languages). Adhering to aspects of the forthcoming QVT standard concrete syntax may also be beneficial.
6. *Expressiveness*: Finally, it should provide expressiveness to support expected domain needs; sufficient expressiveness may be a trade off with respect to ease of use.

The above described requirements are related to qualities of the transformation language. Some pertinent aspects for model to text transformation need to be addressed outside the scope of the language itself, and rather in the architecture of the tools implementing the language. Specifically, this is valid for change management scenarios such as incremental generation, reverse engineering, and round-trip engineering. Support for traceability between model elements and generated text can facilitate these aspects. Traceability links are also independent of the transformation language itself, although the language may open for defining configuration properties that control the nature of traceability links. The language itself may also define mechanisms to control the processing of such links.

The following chapter will look at the MOFScript language in detail and discuss how it meets the identified requirements.

4 The MOFScript Language

The MOFScript language has been defined to answer the needs of a standardized model to text transformation language, as called for by the OMG in the MOF Model to Text Transformation RFP[4]. MOFScript is based on the QVT-Merge[3] specification in terms of metamodel extensions and lexical syntax.

A MOFScript rule is a specialisation of QVT-Merge operational mappings, and MOFScript constructions are specialisations of QVT-Merge constructions. The main goals with the language are to provide ease-of-use, minimize additions to QVT, as well as providing flexible mechanisms for generating text output. It is presumed that a source metamodel is defined on which one can perform queries. This is analogous to QVT, while the explicit definition of a target metamodel is not required in MOF-

Script. MOFScript can be classified as an imperatively oriented language with traditional scope rules and with optionally typed variables.

The following sections look at the details of the MOFScript language, a tool that implements it, and an evaluation considering the requirements identified.

4.1 The Lexical Language

Module: MOFScript transformations are packages within modules, which defines the properties and rules of a transformation. A module is denoted with the keyword "textmodule" followed by a name for the module. The initial part of the module is identical to a QVT mapping rule module except for the keyword which is called *textmodule* as opposed to the QVT *module*.

```
textmodule UML2WSDL (in uml:uml2)
```

A module can import and reuse rules defined in other modules. This is achieved with the 'access library' statement.

```
access library Uml2wsdl ("uml2wsdl-lib.m2t")
```

Rules: The transformation rules are defined with a name and a potential context type. A rule may have a return type. It may also have a guard. The syntax is similar to that of QVT mappings. There is no specific keyword associated with the declaration of rules.

```
uml.Class::classToJava () {
  // statements     }
```

The guard for a rule is defined in the same manner as guards in QVT, using a 'when' clause.

```
uml.Class::classToJava ()
   when {self.getStereotype() = 'Entity'}
{   // statements    }
```

Files and Output Printing: Files are the most important kind of output device for text. A file is declared with a set of properties: name, extension, directory, and type. The File name property must always be present. File name and directory can be specified as separate properties. The directory portion may also be embedded in the file name property.

A file can be used implicitly or explicitly in output statements. For example, if a file device is declared, subsequent output statements will use that device as the target. If several file devices are declared, the latest one is used by default. If a specific device is the target, it can be referenced by its name. Output printing is done by using standard print functions or escaped output. The standard print functions are either 'print' or 'println', which output an expression (the latter adds a new line character (for the appropriate platform / encoding).

A couple of other utility print functions are defined, to provide easier whitespace management: newline (or nl), tab, or space, followed by an optional count integer. Standard String escape characters (\n\t) are also legal within String literals.

```
file ("an-output-file.txt")
<%
   This text is generated to the output file.
%>
```

```
file f2 ("AnotherFile.txt")
file ("Yet-another.txt")
println (" Now, I am writing to the file 'yet-another.txt'")
f2.println (" Now, I am writing to the file 'AnotherFile.txt'");
```

Escaped Output: Escaped output provides a different and in some cases simpler way of providing output to a device. Escaped output works similar to many scripting languages, e.g. Java script.

Escaped output is signalled by escape characters, beginning and ending of an escape. Basically, it is a print statement that can subsume multiple lines and be combined with all expressions that evaluate to a string. Escaped text is signalled by the characters '<%' to start an escape and '%>' to end an escape. Note that whitespace is copied to the output device.

```
uml.Operation::bindingOperation () {
  <%
   <operation name="%> self.name <%">
    <soap:operation soapAction="%> nameSpaceBase + self.name <%"
style="document"/>
     <input>
            <soap:body%>
             if (self.ownedParameter.size() > 0) {
              <% parts="%> self.getParameterOrder() <%"%>
                  }
                  <% use="literal"/>
         </input>
         <output>
          <soap:body%>
           if (self.returnResult.size() > 0){
                  <% parts="response"%>
           }
            <% use="literal"/>
         </output>
   </operation>
  %>
}
```

Properties: Properties are used in the same manner as in QVT. They can be defined at the module level or within a rule. There are two types of properties; local properties which are constants within a module or a rule, and configuration properties which are global properties that may be used in many transformations.

```
property javaPackageName = "org.sintef"
```

A property cannot be modified after its declaration. It is typically used in output statements.

```
<% The Java package name is: %> javaPackageName <% Nothing more,
nothing less %>
```

Variables: Variables are defined and used as in QVT. They can be defined globally for a module, or locally within a rule. A variable can have an assigned value when declared, which can be modified during its lifetime.

```
var exportCounter = 0
var modifiableName:String = "temporary name";
var storedNames:Hashtable;
```

A variable can be typed. The standard OCL types are used (String, Boolean, Integer, Real). In addition, the Collection types List and Hashtable are introduced in MOFScript, which are similar to List and Hashtable classes in Java. These are used for holding sets of values during transformation execution, e.g. to temporarily store pre processed information that is needed several times during generation.

```
var packageNames:List
var packageIdList:Hashtable
self.ownedMember->forEach(p:uml.Package) {
  packageNames.add (p.name)
  packageIdList.put (p.id, p.name)
}
if (packageIdList.size () > 0) {
 <% Listing the package names that does not start with 'S' %>
 packageIdList->forEach (s:String | not(s.startsWith("S")) {
   <% Package: %> s
 }
}
```

Iterators: Iterators in MOFScript are used primarily for iterating collections of model elements from a source model. A for-each block expression defines an iterator expression which also has a block of executable expressions.

It works similarly to forAll in OCL or the shorthand iterator expressions from QVT.

```
-- applies to all objects in the collection
-- of type DBTable that has a name that starts with 'a'
c.elements->forEach(e:DBTable | e.name.startsWith("a")) {
  -- statements
}
```

If-Then-Else: If-expressions provide basic functionality for controlling execution based on logical expressions. An if-expression has a condition and a block of statements that are executed if the condition is met. It might have a set of else conditional branches and an empty else branch. It basically has the same semantics as any conventional programming language if statement.

```
uml.Package::interfacePackage () {
 if (self.name = "Interface Model") {
  self.ownedMember->forEach(p:uml.Package) {
     p.interfacePackages()
  }
 } else {
  stdout.println ("Error in model.")
 }
}
```

Invoking Rules: Text transformation rules are invoked either directly or as part of expressions.

```
uml.Package::interfacePackages () {
 if (self.getStereotype() = "Service"){
  file (rootdir + self.name.toLower() + ".wsdl")
  self.wsdlHeader()
  self.ownedMember->forEach(i:uml.Interface) {
```

```
        i.wsdlPortType()
     }
     self.wsdlFooter()
    }
  }
```

Return Results: Text transformation rules may have return results. This is most useful for defining *helper* functions.

```
uml.TypedElement::getType () : String {
 if (self.type.name.equalsIgnoreCase("string"))
  result = "xsd:string"
 else
  result = self.type.name
}
```

Library Functions: MOFScript defines a set of functions to support manipulation of strings and collections. The string manipulation functionality is similar to that provided in Java. In addition it defines utility functions to manage white space, and functions to retrieve system date and time. It currently does not provide additional functions to interact with the system environment.

4.2 The MOFScript Tool

This section gives an overview of the MOFScript tool, a tool supporting the definition and execution of model to text transformations using the MOFScript language, implemented as an Eclipse plug-in, which is available for download[8].

The architecture: The MOFScript tool is developed as two main logical architectural parts: tool components and service components (see Fig. 1). The tool components are end user tools that provide the editing capabilities and interaction with the services. The services provide capabilities for parsing, checking, and executing the transformation language. The language is represented by a model (the MOFScript model), an Eclipse Modeling Framework (EMF) model populated by the parser. This model is the basis for semantic checking and execution. The MOFScript tool is implemented as an Eclipse plug-in using the EMF plug-in for handling of models and metamodels.

The Service Components consist of these component parts: The *Model Manager* is an EMF-based component which handles management of MOFScript models. The *Parser and Lexer* are responsible for parsing textual definitions of MOFScript transformations, and populating a MOFScript model using the Model Manager. The parser is based on antlr[7]. The *Semantic Checker* provides functionality for checking a transformation's correctness with respect to validity of the rules called, references to metamodel elements, etc. The *Execution Engine* handles the execution of a transformation. It interprets a model and produces an output text, typically to a set of output files. The *Text Synchroniser* handles the traceability between generated text and the original model, aiming to be able to synchronize the text in response to model changes and vice versa.

The Tool Components consist of these component parts: *The Lexical Editor* provides the means of editing transformations, invoking the parser, checker, and the execution engine. The *Result Manager* is responsible for managing the result of a transformation in a sensible way, such as integrating result code files in an Eclipse project.

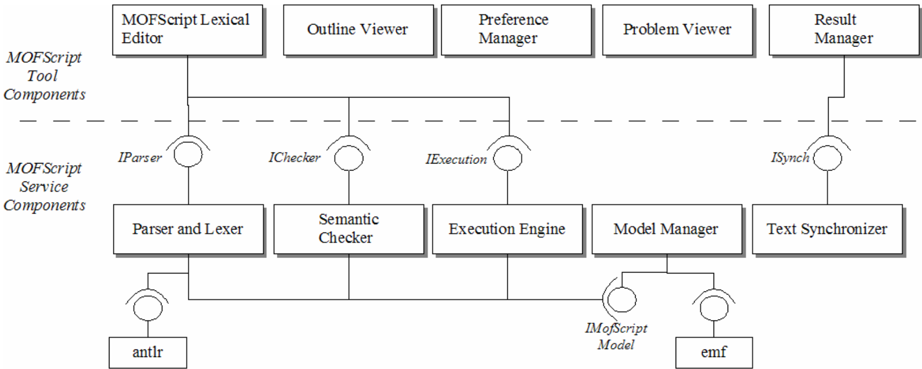

Fig. 1. MOFScript component and tool architecture

The *Outline Viewer*, Preference Manager, and *Problem Viewer* provide simple graphical components to guide the user in writing and executing transformations.

The Model: This section shows the model design used in MOFScript. It is used to generate the EMF model representation of MOFScript, which in turn is utilized by the parser (which produces instances of it) and the Execution Engine. Fig. 2 shows the main MOFScript model structure.

- The *MTTTransformation* class represents a MOFScript transformation module. It has a name, it imports a set of other transformations, it may have parameters, and variable/constant declarations. Finally, it has a set of transformation rules.
- A *TransformationRule* represents a rule (or a function) within a MOFScript transformation. A rule owns a set of statements (*it is a MTTStatementOwner)*, and may have parameters and a return type. Rules define the behaviour of a transformation.
- The *MTTImport* class represents the import of external transformations for a transformation module (MTTTransformation). It is represented by a name and a URI.
- The *VariableDeclaration* class represents variable or constants (properties) for a module (or for statement owners. It has a name, a type, a constant flag and a calculatedValue property (to store the value of simple variables). Basic OCL variable types are supported (String, Boolean, Real, Integer), as well as List and Hashtable types.

A transformation rule consists of different kinds of statements that define the operational logic of the rule:

- The *PrintStatement* class represents printing to a file or to standard output. Print statements produce output towards either the current output device or an explicit prefixed output device. A special syntactic kind of print statement is escape statements, which provide direct output without a print/println command. A print / println statement without prefix will produce output to the current output device. The same will an escaped output statement do.
- The *ResultAssignment* represents the assignment of a value to the result of a rule.

- The *IteratorStatement* represents a loop that iterates on a collection of elements (typically a collection of model elements in a source model). For each element in the collection, a set (a block) of statements is executed.
- The *IfStatement* represents a normal if statement with a condition. It may have an else branch.
- The *GeneralAssignment* represents an assignment of a value to a variable
- The *FunctionCallStatement* represents an explicit call to a rule.
- The *FileStatement* represents the declaration of an output file context, which can be used to print output with print statements.

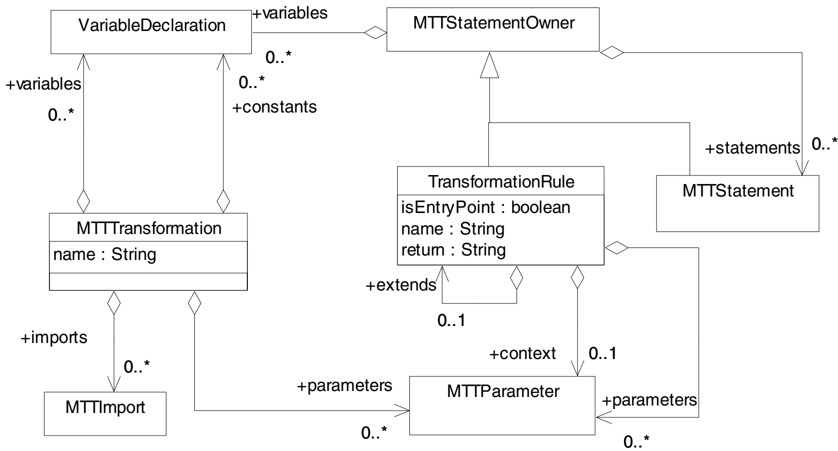

Fig. 2. MOFScript model structure

The User Interface: The MOFScript tool UI is provided through Eclipse editor functionality. It encompasses, as depicted in Fig. 1, a lexical editor, an outline viewer, a configuration manager, and a problem viewer. The lexical editor provides syntax high-lighting and useful code completion associated with the currently active metamodels.

4.3 Change Management

Change management is a highly pertinent issue for model to text generation and involves several aspects, such as management of manual changes to generated code, management of changes to source models, handling reverse engineering and model synchronization, tracing mode information to generated code, and round-trip engineering.

MOFScript does not specify any language-specific mechanisms to support traceability, but a metamodel has been defined to potentially manage the traces from a source model to target files.

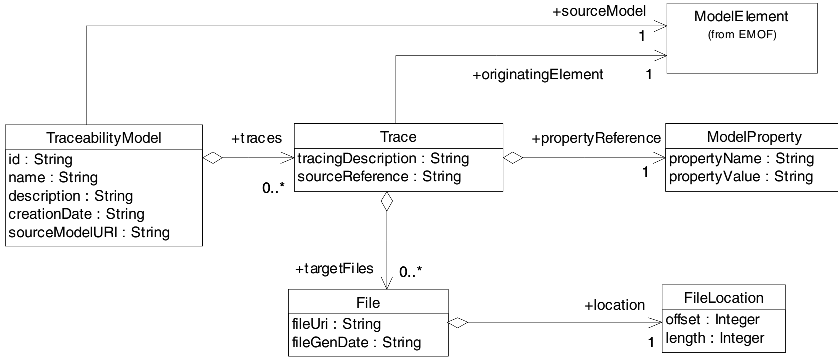

Fig. 3. Model to Text Traceability Model

Traces in this model are managed per model element of the source model. For each (relevant) model element, links are managed to files and file locations within those files that reference the model element.

Source model changes: Changes to source models are only an issue if the already generated text/code has been manually modified, and not yet synchronized with the model. In this case, traceability information can be used to synchronize modified text with newly generated.

Traceability of model information in generated code: In order to support manual changes in generated files, a kind of traceability mechanism that associates generated text with model elements must be in place.

A commonly used solution for handling this is to provide *tags* in the generated code, which establishes the relationship with a part of the text (such as a property or a method) with a model element. This kind of scheme will define a set of relationships between the generated text and the model. These kinds of tags are however dependant of the target language, so they cannot be standardized. The MOFScript language must offer a flexible, user-defined tag mechanism, which can be used as delimiters in the generation (typically, these are embedded as part of comments, Javadoc or similar).

Another solution is to manage traceability information in a separate model, referencing the source model and the generated text files.

4.4 Evaluation of MOFScript

MOFScript supports the basic requirements described in the OMG RFP, i.e. it is capable of generating text based on MOF M2 metamodel specifications, it supports manipulation of strings, it can generate files, etc.

This section looks at the additional requirements and assesses how MOFScript meets those requirements.

1. *Structuring*: Structuring is supported through definition, composition, and invocation of transformation rules. A transformation can import other transformations, and a rule can invoke other rules in a structured manner.

2. *Control mechanisms*: Control mechanisms are provided by supporting iteration over model collections as well as for conditional processing (if-statements).
3. *Mix code and clear text*: Code, clear text output, model references and other expressions can easily be combined in print and escaped print statements.
4. *System services*: MOFScript provides the ability to interact with a limited set of system services, based on what is considered most useful. This is open for future extensions.
5. *Ease-of-use*: The MOFScript language was originally designed to have a look and feel similar to existing programming languages. It then migrated toward the look and feel of the QVT textual concrete syntax in order to establish the compatibility at that level.
6. *Expressiveness*: The expressiveness of MOFScript is kept on an as-simple-as-possible level and defined on a need-to-have basis. Its resulting concrete syntax is therefore quite simple, yet expressive enough to handle complex model to text transformation tasks.

The current MOFScript tool[8] realisation implements all aspects of the language described here, except for guards on transformation rules and change management functionality.

5 Related Work

The most relevant work in this context is the alternative languages submitted to the OMG MOF Model to Text RFP process in three other proposals.

Basically, all the proposals meet the general requirements of the RFP. This chapter will describe the proposals and discuss their positions concerning the additional requirements identified in this paper.

5.1 MOF2Text Partners and the Template Language Specification (TPL)

The MOF2Text partners consist of Mentor Graphics, Pathfinder Solutions, and Compuware Corporation. Their submission [9] presents an imperative approach which also focuses on aspect-oriented concepts. The concrete language is called Template Language Specification (TPL). TPL defines *Patterns*, which are basic structuring mechanisms, similar to modules. They can extend and import other patterns or block libraries. It defines *Methods* as invokable units, which are evaluated in the context of an active output buffer. Methods have parameters, and seem to be defined without any particular metamodel context. A special kind of parameter is a *Literal Parameter*, which allows for sending complex literal expressions as parameters (these may have parameters themselves). File statements declare active output buffers. The language provides basic control statements in terms of ‘if’-statements and ‘for’-statements, and variable assignment in terms of ‘let’-statements.

This submission focuses heavily on the use of aspects as a central mechanism. These are defined within the metamodel, but are not so visible in the concrete syntax.

The listing below gives a brief overview of how this submission meets the identified additional requirements.

1. *Structuring*: Structuring is supported in terms of Patterns and Methods. In addition, the concept flexible literal parameter, allows for complex parameters to be passed to methods.
2. *Control mechanisms*: Control mechanisms are provided in terms of for-statements and if-statements.
3. *Mix code and clear text*: Clear text can be combined with model expressions to produce output.
4. *System services:* The MOF2Text submission defines a set of operations for string and buffer manipulation. It also defines the notation of context operations, which support the notion of functions on well-known objects (e.g. introspection). Additional environment operations are not mentioned.
5. *Ease-of-use*: TPL uses a tagged-based syntax, with square brackets that defines keyword tags in an XML-like manner: `CREATE SCHEMA [SCHEMANAME(schema)/]`; The MOF2Text metamodel is aligned with MOF and OCL, but does not seem to consider the QVT metamodel. The concrete syntax (TPL) is not aligned with QVT. It appears to be less than easy-to-use.
6. *Expressiveness:* The TPL concrete language provides a tag-based syntax, providing advanced template-like substitution mechanism with the literal parameters. There is however discrepancies between the concrete syntax and the metamodel described, e.g. in lack of aspect support. The language seems to provide a high degree of expressive power.

5.2 Interactive Objects (IO)

The IO submission [10] presents a declarative approach with a two-phase transformation strategy. The first phase is the calculation of a target text model based on transformation rules. The second phase is the serialization of text from this model. The submission defines a range of special structuring concepts: Artifact, Section, and Slot define the things to generate. In practice, artefacts represent files. Sections represent (method-like) parts of those artifacts. Slots are properties of an artifact, which are assigned at runtime. An artifact defines parameters (typically with types from the source metamodel) similar to an operation. A Section defines a kind of method which is used by an artifact. It always returns a sequence of its respective type. Pool parameters represent references to collections of objects (typically from the source model), assigned using an OCL expression. The concept Record is used to define functions that cannot produce output text. These are used to group construction of multiple artifacts. The concept Transformation defines an entry point for a transformation, relates to the source metamodels, and invokes artifact sections. The actual text production is performed by templates linked with artifacts and sections. These are defined externally (in separate template files) and provide output text combined with usage of section and artifact slots (properties).

The listing below gives a brief overview of how this submission meets the identified requirements.

1. *Structuring*: The IO language defines a particular kind of language to structure text transformations, controlled by artifacts, sections and slots. These effectively represent model elements of a text/file model, which in turn is used to generate text using text templates.
2. *Control mechanisms*: Control mechanisms are implicit in matching mechanisms of templates. No explicit mechanisms seem to be defined.
3. *Mix code and clear text*: Code and text output are combined within template files.
4. *System services*: The IO language does not specify any ability to interact with system services.
5. *Ease-of-use*: The IO language seems designed to match the artifact metaphor used in IO's tool (ArcStyler). This gives it a distinct structure and style to match the IO graphical transformation structure. It does not seem to reuse any part of the QVT metamodel or syntax. It does, however, reuse OCL for expressions. The language architecture may cater for a high learning curve, but may also be easy to use when first learned.
6. *Expressiveness*: The IO language proposes a declaratively tuned language, where artifact structure is defined independently of template files. Although defining a lot of specialised concepts, it the approach seems flexible and providing for sufficient capabilities.

5.3 Tata Consultancy Services (TCS)

The TCS submission [11] defines an imperative approach based on templates rules. Template rules can be structured into modules. A template rule consists of output text (clear text) in combination with control logic (such as for loops) and metamodel references. Other template rules are invoked explicitly. Template rules may have guards and may override other template rules. A module can extend other modules. Conceptually, the TCS submission seems similar to MOFScript, although different in look and feel.

The listing below gives a brief overview of how this submission meets the identified requirements.

1. *Structuring*: Structuring is provided in terms of modules that can import other modules, and rules that can invoke other rules.
2. *Control mechanism:* Control mechanisms are provided in terms of for-loops and guards on rules.
3. *Mix code and clear text:* Output combines clear text with model reference expressions.
4. *System services:* The TCS language specifies a library for string manipulation and setting current output file. It does not provide other means of system library interaction.
5. *Ease-of-use:* The TCS language defines a quite simple syntax that combines template code with clear text output, similar to a scripting language. The TCS language reuses MOF and OCL concepts, but does not seem to relate to QVT. It appears to be as an easy-to-use language.
6. *Expressiveness:* The TCS language is based on simple principles of templates providing output and calling other templates. It seems to have necessary expressiveness to support complex text transformations.

5.4 Summary

Based on this comparison we can learn that it is not that easy to differentiate the concepts in the different submissions. Although different in the flavour of concrete language, the conceptual differences are not that big. Clearly, concepts such as the aspect-oriented focus of the MOF2Text proposal are clearly distinct. The two-phase transformation focus of the IO proposal appears conceptually distinct in its more declarative approach, as well as the separation between structure and output templates. The TCS proposal is conceptually very close to what is proposed in MOFScript with most distinctions at the concrete syntactical level.

6 Summary and Conclusion

This paper has described the MOFScript language and tool with evaluation against a set of criteria that we see as important for a model to text transformation language standard. These criteria are used also to evaluate the other proposals for the MOF OMG Model to Text transformation RFP.

The MOFScript language and tool allow a user to define model to text transformations from instances of arbitrary metamodels. The language has party based on the definitions from the current QVTMerge specifications, thus keeping the family of transformation languages as similar as possible.

The implementation of the MOFScript tool as an Eclipse plug-in allows for its usage as part of a MDD workbench that can include modelling tools, model to model transformations and model to text transformations in addition to the standard programming environment. We believe that it is necessary to have the model to text transformation tool (at least the execution part) as a tightly integrated part of the MDD tool chain or workbench. Otherwise the number of tool changes needed to complete a full MDD iteration will become too large, causing developers to loose focus through the context changes. This need should of course be balanced with the important issue of choosing the best tool for the task.

Currently the MOFScript tool and language are being used in pilot projects within the MODELWARE[*] project in order to assess the ideas and to provide feedback and input to the further development. Early feedback indicates that some of the QVT like syntax is somewhat unfamiliar to the developers.

From the current status of OMG submissions, it is not easy to see exactly which direction the standard for model to text transformation is headed. A standard needs to accommodate several requirements, but most importantly, it needs to be *usable* and *used*. Time will show if the involved parties are capable of arriving of a best-of-breed integration that will be able to meet this requirement.

Acknowledgment. The work presented here has been carried out within the MODELWARE project (IST Project 511731)[*].

---

[*] MODELWARE is a project co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2002-2006). Information included in this document reflects only the author's views. The European Community is not liable for any use that may be made of the information contained herein.

References

1. Meta Object Facility 2.0, MOF 2.0 Core Final Adopted Specification, OMG document ptc/03-10-04
2. MOF 2.0 Query / Views / Transformations RFP, OMG document ad/2002-04-10
3. QVT-Merge Group, Revised submission for MOF 2.0 Query/Views/Transformations RFP version 2.0, OMG document id ad/2005-03-02, http://www.omg.org/cgi-bin/apps/doc?ad/05-03-02.pdf
4. MOF Model to Text Transformation Language RFP, OMG document ad/04-04-07, http://www.omg.org/cgi-bin/apps/doc?ad/04-04-07.pdf
5. MOFScript Revised Submission to the MOF Model to Text Transformation RFP , OMG document ad/05-05-04, http://www.omg.org/cgi-bin/apps/doc?ad/05-05-04.pdf
6. J. Oldevik, T. Neple, "*Model Abstraction versus Model to Text Transformation*", position paper at European Workshop on MDA (EWMDA)
7. ANTLR, ANother Tool for Language Recognition, http://www.antlr.org/
8. MOFScript Eclipse plug-in, http://www.modelbased.net/mofscript
9. MOF2Text Partners Revised Submission for MOF Model to Text Transformation Language RFP, OMG document ad/2005-05-14, http://www.omg.org/cgi-bin/apps/doc?ad/05-05-14.pdf
10. Interactive Objects MOF Model-To-Text Transformation Language RFP – First Revised Submission, OMG document
11. Tata Consultancy Services, Submission for MOF Model to Text Transformation Language, OMG document ad/2005-05-15, http://www.omg.org/cgi-bin/apps/doc?ad/05-05-15.pdf
12. UML 2.0 OCL Specification (OCL 2.0), OMG Document ptc/03-10-14