A text model - Use your favourite M2M for M2T

Edward D. Willink

Willink Transformations Ltd, Reading, England,
ed_at_willink.me.uk

Abstract. Models provide a disciplined representation of information. Model-to-Model (M2M) transformations convert between model structures. When a more readable representation is required, Model-to-Text (M2T) transformations convert a model structure to a concatenation of character sequences. We ignore the obvious conversion differences and demonstrate that an unmodified M2M tool can be used for M2T. We achieve this with a standard Text model that post-processes the M2M model output to yield formatted text.

Keywords: Model transformation, M2M, M2T, Text model

1 Introduction

For model enthusiasts, everything is a model and every change is a model-to-model transformation. Models are persisted in a form that is suitable for automated use, often involving XMI files or model databases. Unfortunately, in the real world there are many occasions where a practical representation of a model is required. If the information is to be maintained by humans, something more readable than XML is required. If the information is to be consumed by a tool, such as a C compiler that has no knowledge of models, the information must be provided in the tool-specific language, such as C. A model-to-text transformation is therefore required to provide the model in the required form.

An M2T destroys models whereas an M2M preserves models; consequently there is an orthodoxy that M2T and M2M are different technologies. An M2T language is designed around string template expressions, whereas an M2M language is designed around model mappings. This orthodoxy is endorsed by the Object Management Group (OMG) where M2T is specified by MOFM2T [10] and M2M by Query/View/Transformation (QVT) [12]. Acceleo [13] is the only (partial) implementation of MOFM2T. There are a variety of correspondingly named implementations of the QVT languages [15], [16].

In Section 2 we examine the distinctive characteristics of M2T, which we realize using an M2M in Section 3. In Section 4 we evaluate the solution before outlining future work in Section 5. We consider related work in Section 6 and conclude in Section 7.

2 Is M2T necessarily distinct?

At this point I must thank Toni Siljamaki for his obstinate persistence in asking why Eclipse QVTo couldn't be used to perform M2T [7]. My initial instinctive reaction was to fall back on orthodoxy; OMG provide distinct specifications and anyway M2M and M2T are obviously different. However after some reflection, it is clear that the "M2" facilities such as metamodels, loading, navigation, expressions, queries and rules are the same for both M2M and M2T. Only the final "2M" or "2T" need be different. Requiring the use of distinct M2M and M2T tools is not justifiable.

How might an M2M support text? Only one change and one enhancement appear to be necessary:

- change: output to a text file rather than model file
- enhancement: better facilities for creating strings / character sequences

2.1 Text Output

In MOFM2T, the rules (templates) conspire to concatenate and return Strings. M2M tools can already do this since the underlying language, typically OCL [11], has a String type and String operations. It is just necessary to amend the output model declaration to identify that the overall String result is to be emitted directly as a text File, rather than encoded as an XML file.

2.2 Text Facilities

Considering the following example output text:

```
The name is "computedName".
```

Typical M2T tools such as Acceleo or Xtend provide a forward escape to embed a control expression within literal text.

```
The name is "[self.name/]".  -- MOFM2T/Acceleo with escaped OCL
The name is "«this.name»".   -- Xtend with escaped Java
```

A backwards escape may define a multi-line text expression that concatenates with its siblings and may have nested forward escapes.

```
'The name is "[self.name/]".'       -- MOFM2T
'''The name is "«this.name»".'''    -- Xtend
```

An OCL-based M2M without escapes may only concatenate explicitly.

```
'The name is "'+ self.name +'".'    -- OCL
```

OCL provides a fairly modest String library and an overall expression evaluation capability that allows complex String results to be computed. No change is therefore mandated to enable an OCL-based M2M to be used as an M2T although enhanced String capabilities could well be helpful for applications with a high proportion of literal text.

Considering the following example output with a repeated computation:

```
{ "firstComputedName", "secondComputedName" }
```

A for loop facility with optional before/prefix, separator and after/suffix texts is helpful. The examples below show Acceleo and then Xtend.

```
{ [for (p : P | somePs) separator(', ')]"[p.name/]"[/for] }
{ «FOR p : somePs SEPARATOR ', '»"«p.name»"«ENDFOR» }
```

An OCL-based M2M may use collect() for the content iteration, but the separator splicing is verbose.

```
'{ '+ somePs->collect(p | '"'+ p.name +'"')
  ->iterate(s; acc : String = '' |
      if acc = '' then s else acc +', '+ s endif
    )  +' }'
```

3 M2M solution

Figure 1 shows the typical components of an M2M transformation and associated environment that we re-use for M2T. The core M2M transforms between input and output models, each of which conforms to its corresponding metamodel, which in turn conforms to a universal metametamodel such as Ecore. The input model is loaded from an input file by an XMI load facility, and the output model is saved by a corresponding XMI save facility.

After many years reflection, the solution to using an M2M for M2T arose from another longstanding problem; how can an M2M be used to perform an XML2XML transformation?

3.1 XML2XML

The need for an XML2XML transformation arises when there is a need to manipulate some aspect of the XMI serialization of a model in a way that is not accessible to a regular M2M. Typically there may be a need to impose some policy on the xmi:ids that are a serialization rather than model artefact. There is no xmi:id within the model technology space and so any solution that tries to expose an xmi:id is messy.

Bézevin [3] identified distinct model, grammar and XSD technology spaces and the early versions of ATL [14] supported XML2XML transformation through the use of custom injectors and extractors as shown in Figure 2. ATL's weak

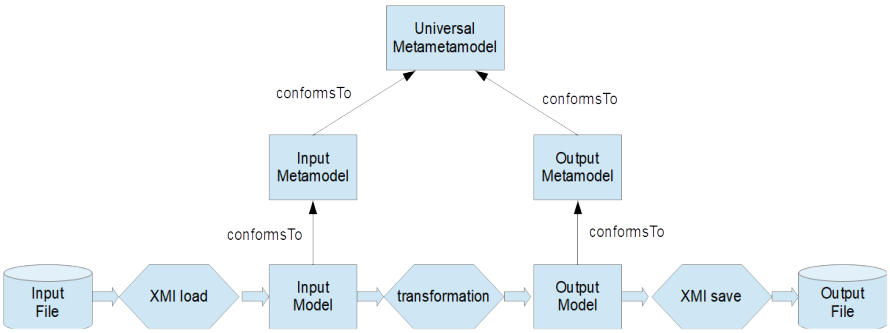

Fig. 1. M2M for models

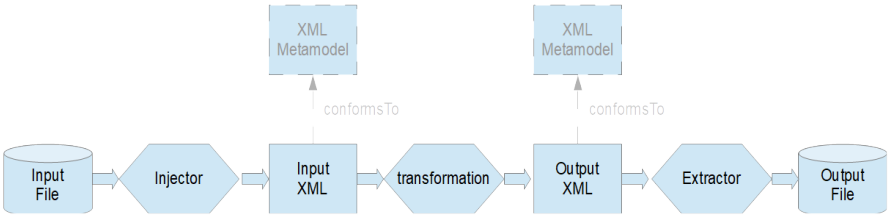

Fig. 2. ATL injector and extractor for XML M2M

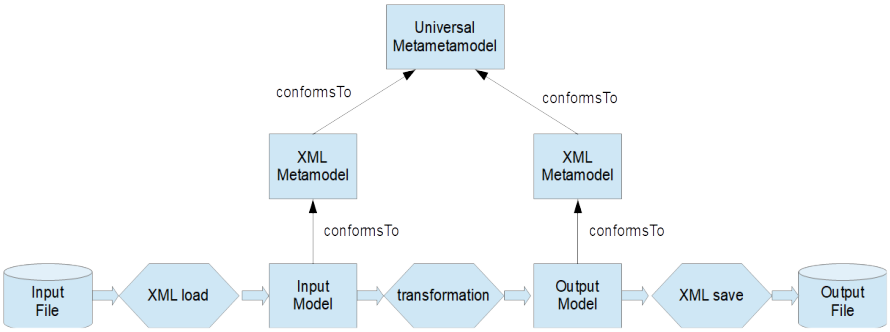

Fig. 3. Variant EMF loader and saver for XML Model M2M

metamodel typing allowed the XML metamodel to remain a little vague. Unfortunately these custom load and save functionalities have suffered from bit rot and are no longer available; XML2XML is no longer supported by ATL.

The proprietary ATL injector and extractor are not necessary when using EMF since the variant loading and saving functionality can be encapsulated by a custom XML Resource allowing it to be used by any M2M tool. We may therefore use the structure shown in Figure 3.

The main difficulty in implementing this approach lies in the absence of a good metamodel for XML. After examining a few candidates, it was clear that none was suitable as a standard against which users could write their XML transformations. The SAX parser [2] is the de facto standard and so the http://www.eclipse.org/qvt/2018/XML metamodel was defined to closely correspond to the familiar SAX parsing events; startElement() creates an Element and starts to populate it; endElement() completes population. The model vocabulary of the XML metamodel is therefore obvious to anyone familiar with the vocabulary of the SAX parser events.

Migrating the functionality from proprietary ATL injector/extractor to EMF Resources enables the XML technology space to be used whenever `*.xmlmodel` is used as the file extension of an input/output model. No modification to tools is required.

3.2 Text Model

The same approach can be used to provide a much simpler and tool-independent solution for using M2M for M2T. This is shown in Figure 4. The M2M transforms to a Text output model which the Text save serializes as a conventional text file.

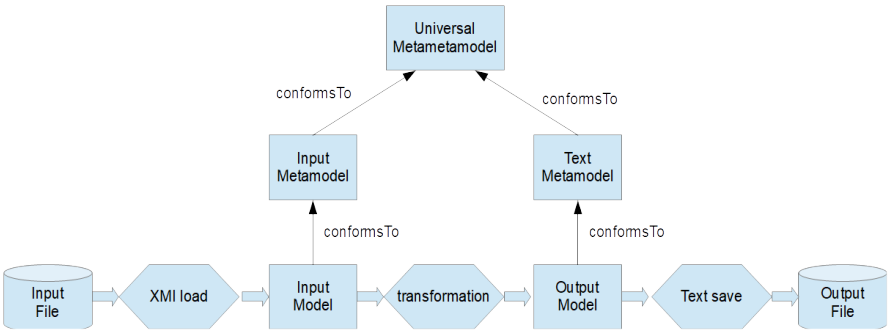

Fig. 4. Use of M2M for M2T

The main challenge is to design a suitable Text metamodel.

3.3 Text Structure

Language parsers such as lex and yacc [1] and their many successors discover the structure of source text by successively aggregating to more powerful concepts:

- 8 bit bytes in a source file/stream
- Multi-byte characters corresponding to Unicode 'letters'
- Multi-character tokens corresponding to 'words'
- Multi-token rules/productions corresponding to 'clauses' and 'sentences'

This conversion sequence is very successful for Text-to-Model conversion but the rules/productions are not appropriate for Model-to-Text conversion since the details of the whitespace formatting are typically discarded once the tokens have been identified. For Model-to-Text purposes, some form of pretty printing is necessary to make the result acceptable to a human reader. We would also like to avoid the language-specific limitations of a typical Text-to-Model grammar.

Traditional M2T tools just concatenate Character Sequences and so we might consider that their `text` metamodel is just a Sequence of Character. Reifying such a metamodel for M2M usage is easy but the need to assign each character as a distinct model element is pretty unacceptable. Perhaps in QVTo:

```
...                                  -- 'The name is'
characters += object text::Character { value := ' '; };
characters += object text::Character { value := '"'; };
forEach (c : self.name.characters()) {
    characters += object text::Character { value := c; };
}
characters += object text::Character { value := '"'; };
```

We should therefore look to some form of multi-Character model.

A significant challenge for practical M2T tools is providing satisfactory indentation and inter-element separation for hierarchical and iterated text structures. The challenge arises because indentation is used within string templates both to indent the output text and to make the control expressions readable.

For Acceleo, the conflict between template and control characters can require empirical iteration to get a satisfactory result. The documentation recommends use of a separate pretty formatting pass if non-trivial formatting is needed.

The Xtend UI uses a grey background to distinguish template characters. This is helpful but not always perfect since the grey background is omitted from new-line characters. Empirical programming cannot always solve the problem.

3.4 Text Metamodel

With a Text metamodel, we can separate the concerns.

- The M2M transformation language statements provide a readable exposition.
- The Text Model declarations control pretty printing.

The Text metamodel shown in Figure 5 compromises a tree of StringNode elements organized by the ordered `parent`/`children` relationship. Additional attributes control the pretty printing performed by the Text Saver.

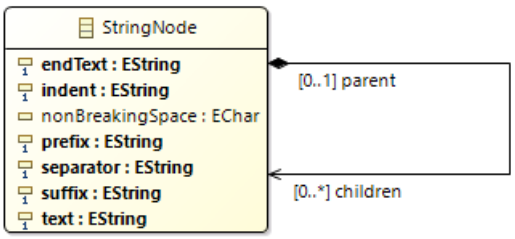

Fig. 5. Metamodel for Text

3.5 Text Saver

The result text is formed as the concatenation of a depth-first traversal of the value of the `text` property of each StringNode.

The detailed formatting of groups of children is facilitated by a `prefix` before the first, a `separator` between each child and a `suffix` after the last child. Additionally an `endText` may follow the children even if there are no children.

The `indent` specifies an indentation increment for this node. The cumulative indentation of a node and its parents starts every non-empty line.

An M2M transforms to create the StringNode tree comprising the important text segments and formatting declarations that influence the final serialization. No characters are provided that are not directed by the StringNode.

The Text Saver is implemented by a custom EMF TextResource that replaces the conventional XML serialization by the pretty printed text resulting from the depth first tree traversal and formatting declarations.

A corresponding Text Loader is also implemented, mainly for test purposes. The serialization during a Text Save is not fully reversible and so the corresponding replacement functionality when loading is limited to identifying whitespace indentation hierarchies and punctuation separators for matching indentations. Any serious Text-to-Model application probably needs a more powerful approach such as an Xtext parser [17].

4 Evaluation

The Text Model has been used in conjunction with Eclipse QVTo to generate C, H, Lex (Flex) and Y (Bison) files from an Ecore file for an auto-generated XMI loader.

No changes were required to Eclipse QVTo and only a couple of minor bug reports were raised. Problems were easily circumvented. Overall there seemed to be fewer problems than when using Acceleo or Xtend. All approaches share the pleasant characteristic of M2T development that problems are clearly visualized in the textual output. Additional debug output can provide helpful insights.

Development time was focussed on the correct content of the Flex and Bison declarations to be generated rather than struggling with their formatting.

Execution of the M2M for text is fast. The 2000 lines of the four XMI loader files are generated in less than a second.

4.1 Example

The utility of this approach can be assessed by examining a very simple example that is artificially elaborated to demonstrate important facilities. The target output is the textile snippet shown in Figure 6. The snippet comprises a heading, a blank line then a list of separated formatted elements.

```
h2(#Precedences). *Precedences*

@NAVIGATION@ > @UNARY@ > @MULTIPLICATIVE@ > @ADDITIVE@ > @RELATIONAL@ > @EQUALITY@ > @AND@ > @OR@ > @XOR@ > @IMPLIES@
```

Fig. 6. Target Textile Output

The Acceleo solution in Figure 7 formats the heading and blank line as literal text at the start of emitPrecedences2. The `[for...]...[/for]` construct supports the formatting of p.name using a `[.../]` escape to access the name. The additional separator argument defines the inter-text separation.

The redundant emitPrecedences1 demonstrates that declarations occur as escapes within the surrounding text literal and the need to re-escape in order to make a nested call.

```
[template public emitPrecedences1(m : Model)]
[emitPrecedences2(m)/]
[/template]

[template public emitPrecedences2(m : Model)]
h2(#Precedences). *Precedences*

[for (p : Precedence | m.getPrecedences()) separator(' > ') ] @[p.name/]@ [/for]
[/template]
```

Fig. 7. Acceleo solution to example

The Xtend solution in Figure 8 is very similar except for the guillemet rather than square bracket escapes.

The redundant emitPreferences1 demonstrates that declarations occur as functions within an outer control flow allowing a direct call to a nested function. The inner emitPrecedences2 uses the `'''...'''` backward escape from the outer

```
def emitPrecedences1(Model m) {
    emitPrecedences2(m)
}

def emitPrecedences2(Model m) {
    '''
    h2(#Precedences). *Precedences*

    «FOR p : getPrecedences(m) SEPARATOR ' > '»@«p.name»@«ENDFOR»
    '''
}
```

Fig. 8. Xtend solution to example

control to the inner text template formatting. This allows the inner template to be indented for readability.

The new QVTo and Text Model solution is shown in Figure 9. It is a little more verbose (or a little more modular). The inner formatting is factored out as emitPrecedence3 that constructs a StringNode comprising just the formatted precedence element. Since there is no string template capability, the self.name expression needs no elaboration, but the text needs quoting and concatenation.

```
mapping Pivot::Model::emitPrecedence1() : Text::StringNode {
    init {
        result := self.map emitPrecedence2();
    }
}

mapping Pivot::Model::emitPrecedence2() : Text::StringNode {
    result.text := 'h2(#Precedences). *Precedences*\n';
    result.prefix := '\n';
    result.separator := ' > ';
    result.children += getPrecedences()->map emitPrecedence3();
    result.suffix := '\n';
}

mapping Pivot::Precedence::emitPrecedence3() : Text::StringNode {
    result.text := '@' + self.name + '@';
}
```

Fig. 9. QVTo and Text solution to example

emitPrecedences2 provides the heading as its text, and iterates emitPrecedences3 to define the children. The separator and children StringNode elements provide very similar capability to the for construct of Acceleo or Xtend.

The redundant emitPrecedences1 demonstrates use of the init section to bypass creation of a nested StringNode. The intermediate Text model instances created by QVTo are shown in Figure 10.

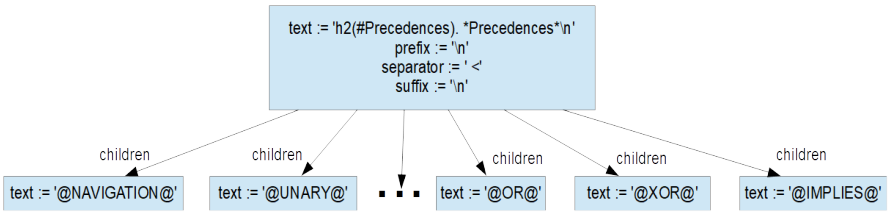

Fig. 10. StringNode instances for the example

4.2 MOFM2T

The OMG issued the MOF Model to Text Transformation Language Request For Proposal (MOFM2T RFP) in April 2007 as a follow on to the MOF 2.0 Query / Views / Transformations RFP (QVT RFP) which addressed only Model-to-Model transformation. The MOFM2T RFP [8] is very clear in its intent:

> This RFP calls for a language and semantics for transforming models into text. The goal is not to create yet another language, but to use or extend existing OMG language(s). Justification must be provided for creating yet another language.

The final MOFM2T submission [9] re-uses the MOF metamodel, OCL expressions and QVT query syntax. However the orthodoxy of the traditional text templates already in use by the submitters' candidates prevailed. The submission fails to re-use any of the transformation facilities of QVT. It is hard to claim that MOFM2T is not 'yet another language'. No justification is provided.

In contrast, in this paper we re-use QVTo (or QVTr or ATL or ...) unchanged to support M2T. This was possible with just one day of development for the one-class metamodel and its associated support.

It might seem that a fully declarative transformation language such as QVTr for M2T could result in the output paragraphs appearing in unpredictable orders? But no, if all collections are ordered, a declarative transformation result should also be ordered; only unordered collections may yield shuffled results.

5 Future Work

The `http://www.eclipse.org/qvt/2018/TextModel` metamodel and its support was made available in June 2018 as part of the Eclipse Photon release once the Eclipse QVTd support is also installed. It may be bundled with ATL and/or QVTo in future releases.

The current support is usable. The text model is powerful, but could be extended. More significant is the opportunity for syntax sugar to make that power more accessible.

5.1 splice iterator

The inelegance of the splicing at the end of Section 2.2 may be mitigated by adding a splice() iterator to the OCL Standard Library for ordered collections of toString()-able elements. This could support:

```
'{ '+ somePs->splice(p; separator=',' | '"'+ p.name +'"') +' }'
```

5.2 Line Wrapping

Line wrapping is not conventionally available for M2T. Separation of information and rendering concerns enables the Text Model saver to respect a prevailing indentation and line-breaking policy and wrap lines accordingly. Figure 5 shows a `nonBreakingSpace` property that is intended to provide a facility to control automatic wrapping of too-long lines. Long lines are first broken at spaces or tabs, then non-breaking space characters are replaced by space characters.

5.3 Declarative Pretty Printing

Tools such as Xtext have demonstrated how the addition of model assignment annotations to an EBNF grammar can synthesize a useful parser/editor rather than just a partial parser. No equivalent extension is available to add pretty-printing annotations, rather Xtext supports manual programming of an independent declarative class. TCS [4] has demonstrated that the required line-wrap/space-before/... declarations can be added to an annotated EBNF grammar. The serializer can then be autogenerated as an M2M text transformation. A friendly editor could offer show-assignment-annotations, and show-formatting-annotations options to avoid redundant clutter.

5.4 String Templates

In the introduction we identified a need to facilitate text synthesis. The evaluation in Section 4 observed that String Templates are not essential, although they may be useful, particularly for output comprising mostly boilerplate.

A string template expression could be added to OCL, in much the same way as it has been to Xtend, but learning from Xtend, Acceleo and MOFM2T, four escape tokens are needed for the start and end of a forward or backward escape. Acceleo has only 2, Xtend and MOFM2T just 3 which inhibits arbitrary nesting. Again learning from Xtend, guillemets are very readable and so we might use:

```
«ocl-expression»        forwards escape within literal text
«'string-template'»     backwards escape - an ocl-expression
```

An OCL expression starting or ending with a ' can be disambiguated with a space. Literal guillemets may be escaped.

```
« 'literal-text' »      disambiguated ocl expression
««»  «»»                escaped literal guillemets
```

These escapes enable the core of our evaluation example from Section 4 to nest an ocl-expression within a string-template within an ocl-expression within a string-template within a QVTo statement.

```
map Pivot::ModelemitPrecedences2() : Text::StringNode {
    «'
    h2(#Precedences). *Precedences*

    «getPrecedences(m)->splice(separator = ', ' | «'@«name»@'»)»
    '»
}
```

An OCL evaluation of string-templates should compute the appropriate string result. An M2M compilation may recognize that string-templates and splice() are syntax sugar for Text Model capabilities.

5.5 Incremental and Parallel M2T

Incremental or parallel execution is difficult in an M2M with imperative characteristics such as QVTo, but very practical in a purely declarative language such as QVTr. Therefore if QVTr is used as the M2M, an incremental execution can mark dirty StringNode elements in the tree that can then be selectively reserialized. Unfortunately incremental QVTr execution is not yet available. A parallel execution can distribute parts of the StringNode tree to multiple proocessors and bring the results together at the end.

6 Related Work

Defining a really simple text model to allow an M2M to be used for an M2T seems like a rather obvious idea. However string template orthodoxy is so entrenched that this simple tree model seems to have been overlooked. Consequently many tools and researchers use naive Strings and then struggle to recover lost structure.

Ogunyomi [5] introduces user-defined signatures to facilitate identifying text segments that need updating. These should be available automatically as a consequence of dependency analysis in a declarative M2M.

Figure 11 shows a Feature Model for our use of M2M for M2T in the style proposed by Rose [6].

This highlights that the mandatory forwards escaping is missing; we have proposed an OCL extension to remedy this. In other respects, the M2M facilities provide good feature coverage and additional features regarding model output. The external text model provides for extensibility.

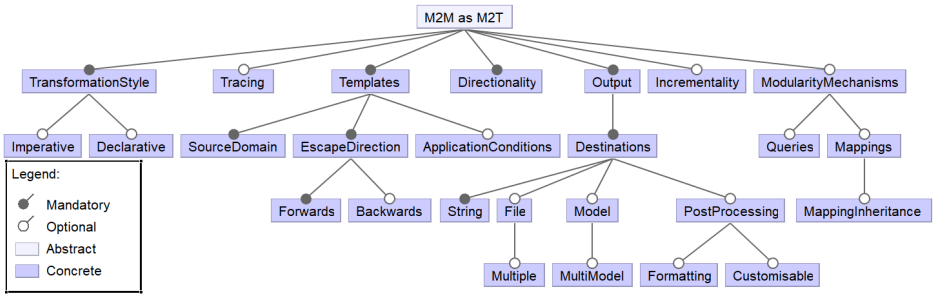

Fig. 11. Feature Model for M2M used for M2T

7 Conclusions

We have introduced a very simple one-class metamodel that models text as a tree of attributed character sequences.

We have shown that this metamodel separates the content and pretty printing concerns and enables any M2M to be used without modification as an M2T.

We have demonstrated the usability of this approach by using Eclipse QVTo to generate c, h, lex and y files from an Ecore metamodel.

We have outlined minor extensions to OCL to improve OCL-based M2M usability for M2T.

We can therefore argue that this simple approach to M2T is more compliant with the OMG MOFM2T RFP than the MOFM2T specification.

References

1. Aho, A., Sethi, R., Ullman, J.: Compilers, Principles, Techniques and Tools, Addison Wesley, 1986
2. Brownell, D.: SAX2, 'Reilly, 2002, ISBN 0-596-00237-8.
3. Ivanov, I., Bézivin, J., Aksit, M.: Technological Spaces: An Initial Appraisal. 1-6. (2002).
4. Jouault, F., Bézivin, J., Kurtev. I.: TCS: A DSL for the specification of textual concrete syntaxes in model engineering. Model-Driven Engineering Languages and Systems: 17th International Conference, MODELS 2014, , September 28 – October 3, 2014, Valencia, Spain. https://research.utwente.nl/en/publications/tcsa-dsl-for-the-specification-of-textual-concrete-syntaxes-in-mo
5. Ogunyomi, B., Rose, L., Kolovos, D.: User-defined Signatures for Source Incremental Model-to-text Transformation. 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006. http://ceur-ws.org/Vol-1331/p4.pdf
6. Rose, L., Matragkas, N., Kolovos, D., Paige, R.: A feature model for model-to-text transformation languages. In Modeling in Software Engineering (MISE), 2012 ICSE Workshop on (pp. 57-63). IEEE. DOI: 10.1109/MISE.2012.6226015
7. Siljamaki, T.: Additional M2T capability in QVTO. Eclipse QVTo project Bugzilla 396543. December, 2012. https://bugs.eclipse.org/396543

8. MOF Model to Text Transformation Language Request For Proposal. OMG Document: ad/04-04-07, April 2004. https://www.omg.org/cgi-bin/doc?ad/04-04-07.pdf
9. Revised submission for MOF Model to Text Transformation Language RFP. OMG Document: ad/06-09-03. September 2006. https://www.omg.org/cgi-bin/doc?ad/06-09-03.pdf
10. MOF Model to Text Transformation Language, v1.0, OMG Document Number: formal/2008-01-16, Object Management Group (2008), http://www.omg.org/spec/MOFM2T/1.0
11. Object Constraint Language. Version 2.4., OMG Document Number: formal/2014-02-03, Object Management Group (2009), http://www.omg.org/spec/OCL/2.4
12. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.3. OMG Document Number: ptc/16-06-03, June 2016.
13. Eclipse Acceleo Project.
https://projects.eclipse.org/projects/modeling.m2t.acceleo
14. Eclipse ATL Project.
https://projects.eclipse.org/projects/modeling.mmt.atl
15. Eclipse QVT Declarative Project.
https://projects.eclipse.org/projects/modeling.mmt.qvtd
16. Eclipse QVT Operational Mappings Project.
https://projects.eclipse.org/projects/modeling.mmt.qvto
17. Eclipse Xtext Project.
https://projects.eclipse.org/projects/modeling.tmf.xtext