A Feature Model for Model-to-Text Transformation Languages

Louis M. Rose, Nicholas Matragkas, Dimitrios S. Kolovos, and Richard F. Paige
Department of Computer Science
University of York
Deramore Lane, Heslington, York, YO10 5GH, UK
{*louis,nikos,dkolovos,paige*}*@cs.york.ac.uk*

***Abstract*—Model-to-text (M2T) transformation is an important model management operation, as it is used to implement code and documentation generation; model serialisation (enabling model interchange); and model visualisation and exploration. Despite the creation of the MOF Model-To-Text Transformation Language (MOFM2T) in 2008, many very different M2T languages exist today. Because there is little interoperability between M2T languages and rewriting an existing M2T transformation in a new language is costly, developers face a difficult choice when selecting a M2T language. In this paper, we use domain analysis to identify a preliminary feature model for M2T languages. We demonstrate the appropriateness of the feature model by describing two different M2T languages, and discuss potential applications for a tool-supported and model-driven approach to describing the features of M2T languages.**

I. Introduction

Model transformation has been characterised as the heart and soul of Model-Driven Engineering (MDE) [1]. An important type of model transformation is model-to-text (M2T) transformation, which is used to implement code and documentation generation, model serialisation (enabling model interchange), and model visualisation and exploration.

Today, developers wishing to implement a M2T transformation face a difficult decision, as there are numerous M2T languages with features that vary considerably. For example, some languages focus on providing tight integration with – and sophisticated developer tools for – a small number of target languages, such as Microsoft T4 [1] which targets .NET languages. Other M2T languages seek not to constrain the form of generated text, but rather to be small and easy to learn, such as Java Emitter Templates[2] (JET).

The difficulty in choosing a M2T language is compounded by a lack of interoperability between M2T languages, which means that it is generally very time-consuming to rewrite an existing transformation to work with a different M2T language. Although the Object Management Group produced a M2T standard in 2008 (MOF Model-To-Text Transformation Language, MOFM2T), only Acceleo[3] seeks to conform to the standard, and other M2T languages deviate from it in both syntax and features. Promisingly however, most M2T languages are template-based [2] (rather than using a visitor pattern or explicit printing statements), and we focus on template-based M2T languages in our feature model and throughout the remainder of the paper.

In this paper, we present preliminary work that seeks to aid developers in selecting a M2T language. The main contribution is a feature model that can be used to distinguish between M2T languages (Section II). The feature model is inspired by previous work by Czarnecki and Helsen [2] on surveying model transformation, but focuses on template-based M2T languages (which Czarnecki and Helsen considered only as a degenerate case of model-to-model transformation). We demonstrate the applicability of our feature model by using it to illustrate the features of the MOFM2T standard and a contemporary M2T language (Section III).

We have constructed the feature model in Section II in a principled manner, using MDE tools and techniques. For example, model validation [3] has been applied to construct domain-specific tool support for instantiating the feature model (Section III). The secondary contribution of the paper is along these lines; in Section IV we present the ways in which we anticipate that the feature model in Section II might be used – along with appropriate model management operations – to provide tool support that, for example, assists in the selection and comparison of M2T languages. As such, we argue that the feature model we present is not simply a static artefact, but a dynamic one that supports manipulation and behaviour.

II. A feature model for M2T languages

Feature modelling is one way in which a system can be decomposed to represent different (often independent) features. Unlike the modular decomposition of an implemented system (e.g., into classes and packages), a feature model should be understandable by – and useful to – users.

In the remainder of this section, we present a preliminary feature model for *template-based M2T languages*. We draw inspiration from a previous feature-based survey of model transformation languages by Czarnecki and Helsen [2]. Like Czarnecki and Helsen, we have sought to categorise the various features of model transformation languages. However, our feature model is significantly different to

[1] http://msdn.microsoft.com/en-us/library/bb126445.aspx
[2] http://www.eclipse.org/emft/projects/jet/
[3] http://www.eclipse.org/acceleo/

978-1-4673-1757-3/12/$31.00 © 2012 IEEE

the feature model of Czarnecki and Helsen [2] for two reasons. Firstly, the latter places a significant focus on *transformation rules* (three of the eight high-level features relate to transformation rules). However, templates – and not transformation rules – are the core of template-based M2T languages. Secondly, Czarnecki and Helsen regard templates as "a degenerate form of [transformation] rules", and, perhaps as a consequence, their feature model is not granular enough to distinguish between template-based M2T languages and does not consider concepts specific to model-to-text transformation (such as how the resulting text is sent to its destination).

The feature model presented in this section focuses on distinguishing between template-based M2T languages, and is an extension to – and re-working of – the feature model of Czarnecki and Helsen. An obvious limitation of the feature model presented in this section is that it is specific to template-based M2T languages, and could not be used for comparing M2T and M2M transformation languages.

A. Approach

The feature model presented in this section was constructed by first comparing six contemporary M2T languages: Acceleo[4], the Epsilon Generation Language (EGL) [4], Java Emitter Templates (JET), Microsoft Text Template Transformation Toolkit (T4), MOFScript [5], and Xtend 2[5]. These languages were chosen as they have been recently updated (i.e., a new release in the past two years) and represent different approaches to development, including commercial (e.g., T4), reference implementations of a standard (e.g., Acceleo), and academic research (e.g., MOFScript, EGL). Comparison involved applying each language to specify an example of M2T transformation (generating Java code from an Ecore metamodel) and by examining the user documentation of each language.

Following the comparison, we listed the primary features of each language, grouping them into categories based on similarities, using the MOFM2T standard and the feature model of Czarnecki and Helsen [2] for guidance. Where appropriate, we have re-used parts of the latter, as described below. Finally, we constructed a feature model and performed some refactoring with the FeatureIDE plug-in for Eclipse [6]. Constructing our models with FeatureIDE facilitates a model-driven approach to managing and manipulating the models and enables the potential use cases described in Section IV. We anticipate further revisions to the feature model following discussions subsequent to the publication of this preliminary work.

B. Notation

The diagrams presented in this section use the notation of FeatureIDE; a legend is shown in Figure 1. In addition, we use a shaded feature to mean a feature specific to model-to-*text* transformation languages, and a non-shaded feature to mean a feature that also applies to model-to-*model* transformation languages and has been re-used from the feature model of Czarnecki and Helsen [2].

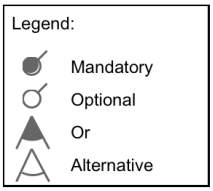

Figure 1. The notation used to represent features.

Note that the use of an *Alternative* to group features denotes that one or more of those features must be implemented; while the use of an *Or* denotes that exactly one of those features must be implemented.

C. Top-Level Features

We now briefly describe the top-level features that we have identified as integral to a M2T language (Figure 2). The features that are not part of the feature model of Czarnecki and Helsen and contain many sub-features are discussed in more detail in the remainder of this section.

1) TransformationStyle: indicates the way in which the flow of control through the transformation is specified, and comprises two mutually exclusive sub-features. *Imperative* languages typically have an explicit entry point, such as a main template or module as is the case in EGL. In *Declarative* languages, the transformation engine schedules the flow of control through the transformation by, for example, non-deterministically selecting source model elements and applying matching templates. This feature is a more restrictive version of the *RuleSelection* feature of Czarnecki and Helsen [2].

2) Templates (Section II-D): are the fundamental units of a template-based M2T language, and this feature relates to the way in which a transformation is decomposed into these fundamental units. This feature is analogous to the *TransformationRules* feature of Czarnecki and Helsen [2]. However, there are variation points in *TransformationRules* that did notapply to *Templates* for the languages that we have examined, such as multi-directionality (none of the languages allows its templates to be run in reverse). Conversely, there are variation points for *Templates* that do not apply for *TransformationRules*, such as the direction in which a template is escaped (i.e., whether code or text is made explicit). The M2T languages considered for the domain analysis are typed and textual, but we note that further investigation might uncover untyped or graphical M2T languages.

[4]http://www.eclipse.org/acceleo/
[5]http://www.eclipse.org/xtend/

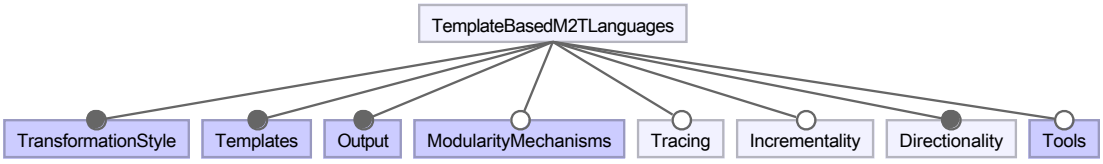

Figure 2. Top-level features

3) Output (Section II-E): refers to the ways in which text may be generated. Languages vary in the *Destinations* to which text can be generated, and the types of *PostProcessing* that may be performed on generated text.

4) ModularityMechanisms (Section II-F): describes the language constructs used to encapsulate re-usable pieces of transformation logic. In an extension to the *ModularityMechanisms* feature of Czarnecki and Helsen [2], we recognise that there are a range of modularity mechanisms available in contemporary M2T languages, including *Queries* (i.e., derived attributes) and *Macros* (i.e., user-defined blocks).

5) Tracing: relates to the ability to describe and navigate between source model(s) and generated text. This feature is re-used from the feature model of Czarnecki and Helsen [2]. Briefly, Czarnecki and Helsen state that the creation of trace links can be either *Manual* (e.g., the user specifies trace blocks in their transformation) or *Automatic* (e.g., the transformation engine creates trace links whenever a model element is accessed during the transformation). Automatic trace link creation might also be *Tunable* (e.g., the user includes metadata in the transformation that is used by the transformation engine to customise automatically created trace links). Finally, trace links might be stored in the *SourceModel*, *GeneratedText*, or *Separately*.

6) Incrementality: describes the way in which a transformation is executed in response to updates to the source model(s) or generated text. This feature is re-used from the feature model of Czarnecki and Helsen [2]. Czarnecki and Helsen identify one mandatory and two optional subfeatures of incrementality: *TargetIncrementality* is the mandatory feature and relates to the execution of a transformation to propagate changes from source model(s) to generated text by, for example, using traceability links; *SourceIncrementality* is analogous to incremental compilation and uses an impact analysis of the source model(s) to minimise the processing required to achieve target incrementality by, for example, reducing the number of traceability links that need to be examined to propagate changes; *PreservationOfUserEditsInTheTarget* is concerned with ensuring that sections of the generated text that have been changed manually are preserved when a transformation is re-run.

7) Directionality: refers to the ability to run a transformation in more than one direction. This feature is re-used from the feature model of Czarnecki and Helsen [2]. Czarnecki and Helsen note that transformation languages must support *Uni-directional* transformations (and, in the case of M2T transformations, this refers to the transformation of source model(s) into generated text), but can also support *Multi-directional* transformations in which a single transformation can be used to specify a forwards and a backwards transformation. None of the M2T languages considered in our comparison supported multi-directional transformations (i.e., a single transformation specification that can be used for M2T and T2M transformation), but a broader comparison might include T2M languages, such as Xtext[6], which do provide limited support for multi-directional transformation.

8) Tools (Section II-F): comprises features that relate to the tool-support provided by a M2T language. We focus on tools that are specific to M2T transformation, rather than general-purpose tools such as transformation editors (with syntax highlighting and auto-completion) and interactive debuggers.

D. Templates

Templates are the fundamental units of a M2T transformation. In Acceleo and MOFScript, transformations are typically specified in terms of several templates that relate to different parts of the source model(s). The *Templates* feature comprises three sub-features (Figure 3).

1) SourceDomain: describes the way in which a language allows the specification of the elements of the source model(s) on which a template will be executed. In Acceleo, for example, each template specifies a parameter of a type contained in the source metamodel. In JET2, a single untyped argument is used to specify the root source model element on which the transformation is to be executed. Czarnecki and Helsen provide a comprehensive discussion of the types of domain used by model transformation languages, and we consequently re-use their *Domain* feature [2] here.

2) EscapeDirection: indicates the extent to which a user can control whether the generated text or the dynamic blocks of a transformation should be escaped. Typically, the intention of template-based specification of M2T transformation is to make explicit the generated text, and hence dynamic blocks are escaped (e.g., `Hello [p.name/]!`). This is

[6]http://www.eclipse.org/Xtext

Forwards escaping. In some cases, however, a template might include many more dynamic blocks than regions of static text. In these cases, some languages support an alternative syntax in which the generated text is escaped, and the dynamic blocks are not (e.g., `'Hello 'p.name'!'`). This is *Backwards* escaping. Only the M2T standard, MOFM2T, supports backwards escaping of templates. Notably, Acceleo (the only reference implementation of MOFM2T) does not currently support backwards escaping.

3) ApplicationConditions: relate to language features that are used to specify conditions that restrict when a template can be applied. In Acceleo, for example, templates can specify a guard part. Note that this feature is identical to the *ApplicationConditions* feature of *TransformationRules* identified by Czarnecki and Helsen [2].

E. Output

Every M2T language must be able to generate text. However, the destinations to which text may be generated and the extent to which the generated text can be post-processed varies between languages. Consequently, the *Output* feature comprises two sub-features (Figure 4).

1) Destinations: indicates the types of target to which a transformation can generate text. Every M2T language must at least be able to generate a *String*, for example, to the standard output stream. Most M2T languages provide support for specifying that text should be generated to a *File* on disk. Generating a file is sometimes restricted to a *Single* file, for example, in the case of Microsoft T4's design-time templates, which emit text to a file named similarly to the name of the transformation file. More often, text can be generated to *Multiple* files, by using dedicated `file` blocks (e.g., MOFScript and Acceleo), or by using features of the host language (e.g., Xtend2). Finally, destination types can be *Extensible* in which a user can add custom destinations. For example, EGL allows users to plug-in custom logic for directing generated text; this functionality has been used to implement web-based model querying and navigation [7], and aspect-oriented code generators [8].

2) PostProcessing: describes the ways in which users can specify additional phases of processing before generated text is sent to its destination. Numerous forms of post-processor are imaginable, but the M2T languages available today provide only three. *Formatting* is used to adjust the appearance of generated text (e.g., by changing the indentation of generated code). The way in which formatting is specified appears to differ for each M2T language: in Acceleo a `post` part of each template can be used to specify text formatting; Xtend 2 provides implicit whitespace correction; and EGL provides several out-of-the-box, standalone formatters for popular target languages. *Weaving* is the process of merging together related portions of generated text emitted by distinct parts of the M2T transformation. For example, a M2T transformation that produces executable code might be decomposed into several templates: one for generating the body of methods, one for generating logging logic, and one for generating method documentation. In this case, the three templates generate related code that must be merged together before it is sent to its destination. Some M2T languages provide *Customisable* post-processing of templates, in which users can extend the language with additional post-processors. For example, EGL allows users to plug-in additional formatters and general-purpose processors to, for example, perform impact analysis on generated code.

F. ModularityMechanisms

The language constructs provided to encapsulate and re-use transformation logic vary between M2T languages (Figure 5). We identify four distinct features along these lines:

1) Macros: are user-defined operations used to define additional types of block. Unlike *Queries*, *Macros* take a formal parameter to which execution can be yielded and hence support nested blocks[7].

2) Queries: are used to encapsulate the complicated navigation of source models. In some cases, they are used to specify derived attributes that do not exist as part of the source metamodel(s).

3) LanguageExtension: allows users to define their own language constructs which can be plugged-in to the M2T language. In JET2, language extensions can be used to implement new tags (blocks).

4) TemplateInheritance: is used to share templates between M2T transformations. Typically, this is implemented with two mechanisms: one for importing templates from other M2T transformations, and another for specifying which templates in the imported or importing M2T transformation should be overriden. MOFScript allows inheritance at the level of M2T transformations, and Acceleo at the level of individual templates.

G. Tools

Tools are optional features provided to ease the development of M2T transformations. Our feature model focuses on tools specific to the domain of M2T transformation (Figure 6), and excludes more general-purpose programming tools, such as syntax highlighting and interactive debuggers.

1) ModelToTextNavigation: provides users with the capability to identify the text generated by a transformation and to navigate between generated text, source model(s) and, in some cases, the M2T transformation. For example, Acceleo provides support for navigating between generated Java code and source model elements. By comparison, EGL provides limited model-to-text navigation allowing users to

[7]In other words, the last argument to a MOFM2T macro definition is a closure [9].

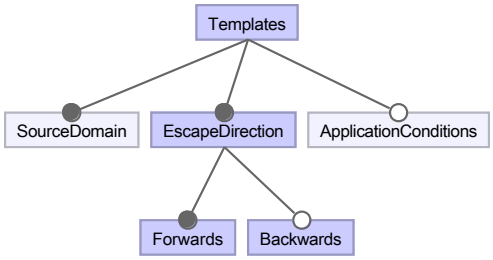

Figure 3. Decomposition of the *Templates* feature.

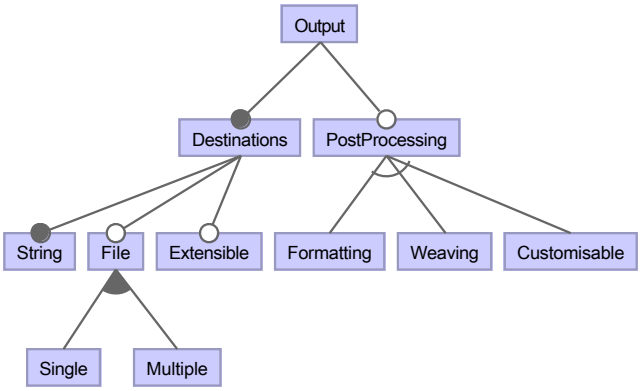

Figure 4. Decomposition of the *Output* feature.

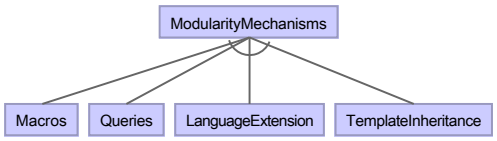

Figure 5. Decomposition of the *ModularityMechanisms* feature.

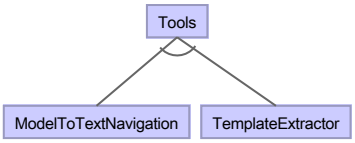

Figure 6. Decomposition of the *Tools* feature.

determine which templates have been involved in a model-to-text transformation, and which files have been generated by each template.

2) TemplateExtractor: is a tool for rapidly developing a M2T transformation from examples of the text to be generated. An example is successively refined into a template by incrementally converting parts of the generated text into dynamic blocks that emit data from source model(s). The rapid text replacement feature of Acceleo[8] allows users to extract templates from existing Java code.

III. Instantiation

To demonstrate the usefulness and appropriateness of the feature model in Section II, we now examine the way in which two M2T languages can be represented with the feature model. The process of representing the features of a M2T language with the feature model is a form of instantiation that has the domain-specific semantics described below.

A. Semantics of Instantiation

An instantiation of the feature model in Section II can be used to represent the particular features supported by a specific M2T language. Therefore, instantiation is mostly concerned with replacing optionality in the feature model with single features. In constructing the instantiated feature models presented in this section, the constraints below have been formulated. When the constraints are satisfied, the instantiated model, `i`, is a valid instance of the feature model, `f`.

- Every feature in `i` must appear in `f` (i.e., instantiation cannot introduce new features)
- Every mandatory feature of `f` must appear in `i` (i.e., instances must include mandatory features).
- For every set of features grouped by an *Alternative* in `f`, one or more of those features must appear in `i`.
- For every set of features grouped by an *Or* in `f`, exactly one of those features must appear in `i`.

The above constraints have been implemented in the Epsilon Validation Language (EVL) [3] and used to check the instantiation of the models presented in the remainder of this section. This is an example of using a principled and tool-supported approach to feature modelling, which we discuss further in Sections IV and V.

B. The Features of MOFM2T

MOF Model-To-Text Transformation Language (MOFM2T) [10] is the OMG standard for M2T transformation. The only version of the standard was published in 2008, which outlines language features,

[8]http://www.obeonetwork.com/page/acceleo-user-guide#HRapidTextReplacement

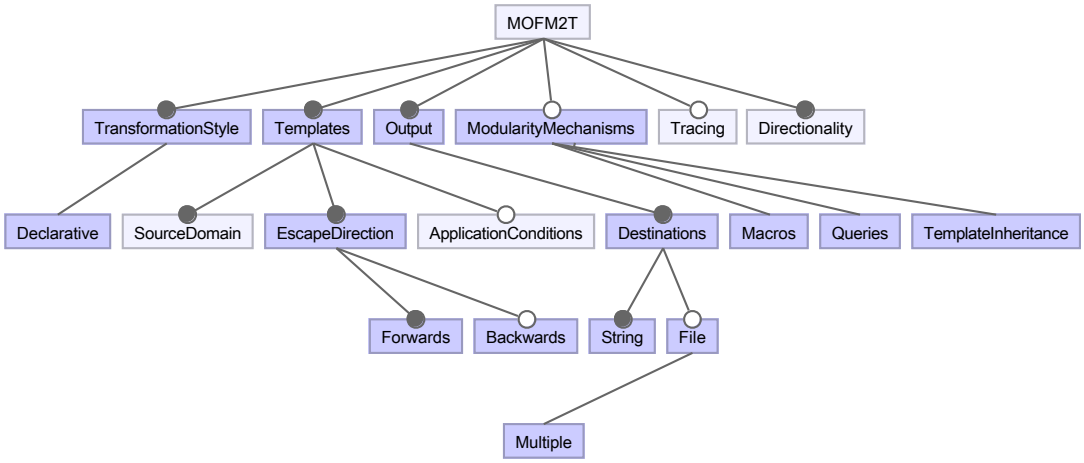

Figure 7. Features of the MOFM2T standard for M2T.

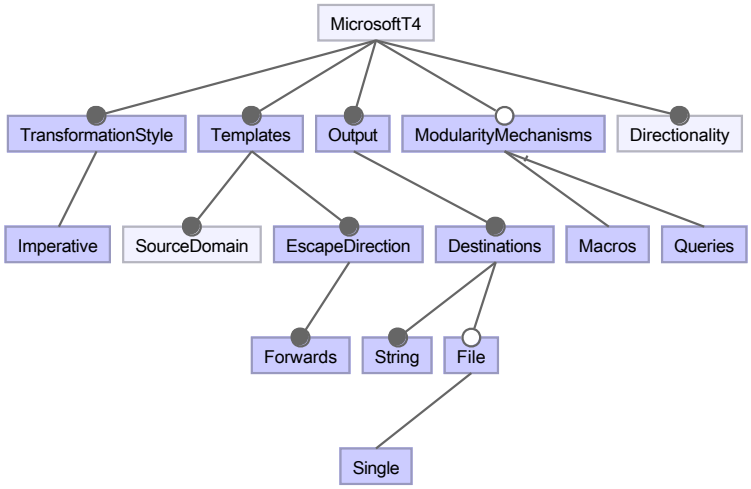

Figure 8. Features of the Microsoft T4 M2T language.

concrete and abstract syntax for specifying M2T transformation.

The result of instantiating the feature model to represent MOFM2T is shown in Figure 7. Instantiating the feature model for MOFM2T demonstrates notable features of the standard, such as support for both *Forwards* and *Backwards EscapeDirections*. By comparing the instantiated model (Figure 7) with the top-level feature model (Figure 2), we can see that MOFM2T provides no support for *Incrementality* or *Tools*.

C. The Features of Microsoft T4

Microsoft Text Template Transformation Toolkit[9] (T4) is part of Microsoft Visual Studio, and provides a language for defining run-time and design-time templates. The former are used to specify the way in which an application should produce textual output, while the latter are used to generate the .NET code used to implement an application. T4 was first released in 2005 and was most recently updated in 2010.

The results of instantiating the feature model to represent Microsoft T4 is shown in Figure 8. The instantiated model demonstrates that Microsoft T4 implements very few optional features; there is no support for *Incrementality*, *Tools*, *Tracing* or *ApplicationConditions* for *Templates*.

[9]http://msdn.microsoft.com/en-us/library/bb126445.aspx

IV. Potential Applications

As demonstrated in the previous section, instantiation of the feature model yields models that can be used to describe a particular M2T language. We envisage several use cases

for the feature model and instantiated models, as described in this section.

A. Determining Compatibility and Conformance

M2T languages continue to evolve, and different versions of a language might support different features. Furthermore, different M2T languages support different features. As new versions of a M2T language are released and as some M2T languages become obsolete or unsupported, we anticipate a need for comparing languages to assess compatibility. A developer, for example, might wish to migrate their M2T transformation to another language. Model comparison between the source language and possible target languages will inform decisions about whether migration is possible. Similarly, model comparison can be applied to check the conformance of a language to a standard, such as MOFM2T.

B. Assessing the Importance of M2T Languages Features

The feature model provides a common vocabulary for discussing the features of M2T languages, and therefore for assessing the extent to which each feature is implemented (i.e., how *widespread* a feature is). Additional forms of analyses might use the feature model to determine the popularity of a feature from the perspective of users. We discuss this further in Section V.

C. Guiding M2T Language Selection

As discussed in Section I, the variety in M2T languages makes it difficult for developers to make an appropriate selection. By constructing a comprehensive set of instantiated models for M2T languages, we anticipate the capability to generate a flowchart for assisting in decision making. For example, the sub-features of *TransformationStyle* and *File* in the instantiated models in Figures 7 and 8 show that MOFM2T and Microsoft T4 differ in these regards. The distinguishing features of a set of instances could be used to determine questions to ask a developer in order to minimise the set of languages that the developer will need to investigate when he or she is selecting a M2T language.

V. Conclusions

We have presented a preliminary feature model for M2T languages. The feature model has been synthesised from a comparison of six contemporary M2T languages and, in future work, we anticipate checking and evaluating the feature model by attempting to describe the features of further M2T transformation languages and of T2M transformation languages, such as Xtext. With the feature model presented in this paper, we seek to encourage further discussion of the features that are necessary for specifying M2T transformation; to promote debate and collaboration between the developers of M2T languages; and to reflect on the appropriateness of the current version of the MOFM2T standard.

In future work, we will continue to refine the feature model based on initial discussion with MDE experts and practitioners. In particular, we will collect and analyse data to determine the relative popularity of the various features of M2T languages. Furthermore, we will further investigate the need for additional structures and processes for *principled and tool-supported* feature modelling. We present our first steps in this direction in Section III, which demonstrates one way in which model management operations might be used during a feature modelling process.

The feature model, semantics for instantiation and possible applications presented in this paper are our first steps towards assisting developers in selecting a M2T language, and we believe that further work in this direction will contribute towards the increased adoption of MDE tools and techniques.

References

[1] S. Sendall and W. Kozaczynski, "Model transformation: the heart and soul of model-driven software development," *IEEE Software*, vol. 20, pp. 42–45, 2003.

[2] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–646, 2006.

[3] D. Kolovos, R. Paige, and F. Polack, "On the evolution of OCL for capturing structural constraints in modelling languages," in *Rigorous Methods for Software Construction and Analysis*, ser. LNCS, vol. 5115. Springer, 2009, pp. 204–218.

[4] L. Rose, R. Paige, D. Kolovos, and F. Polack, "The Epsilon Generation Language," in *Proc. ECMFA*, ser. Lecture Notes in Computer Science, vol. 5095. Springer, 2008, pp. 1–16.

[5] J. Oldevik, T. Neple, R. Grønmo, J. Aagedal, and A. Berre, "Toward standardised model to text transformations," in *Proc. ECMDA-FA*, ser. LNCS, vol. 3748. Springer, 2005, pp. 239–253.

[6] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel, "FeatureIDE: A tool framework for feature-oriented software development," in *Proc. ICSE*. IEEE, 2009, pp. 611–614.

[7] D. Clowes, D. Kolovos, C. Holmes, L. Rose, R. Paige, J. Johnson, R. Dawson, and S. Probets, "A reflective approach to model-driven web engineering," in *Proc. ECMFA*, ser. LNCS, vol. 6138. Springer, 2010, pp. 62–73.

[8] S. Zschaler and A. Rashid, "Symmetric language-aware aspects for modular code generators," Kings College, Department of Informatics, Technical Report No. TR-11-01, Tech. Rep., 2011.

[9] H. Abelson and G. Sussman, *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 1996.

[10] OMG, "MOF Model to Text Transformation Language, v1.0 [online]," [Accessed 17 January 2012] Available at: http://www.omg.org/spec/MOFM2T/1.0/, 2008.