Templatized Model Transformations: Enabling Reuse in Model Transformations

Amogh Kavimandan[1], Aniruddha Gokhale[1⋆], Gabor Karsai[1], and Jeff Gray[2]

[1] ISIS, Dept. of EECS, Vanderbilt University, Nashville, TN, USA
[2] Dept of CIS, University of Alabama at Birmingham, Birmingham, AL, USA

Abstract. Model transformations are a key element of model-based software development processes. Despite their importance, existing model transformation tools and processes have limited support for reuse, particularly, in the context of product line development that must handle variability among product variants. This forces developers to reinvent the transformation rules thereby adversely impacting their productivity and increasing maintenance costs. This paper presents MTS (Model-transformation Templatization and Specialization), which overcomes these limitations by enabling developers to write reusable, templatized model transformations. MTS defines two higher order transformations to capture the variability and to specialize the transformations across variants of an application family. MTS can be realized within existing model transformation tools with minimal modifications. A qualitative evaluation of MTS is presented describing the reduction in efforts to define model transformation rules as new variants are added to the product line.
Keywords: Model transformations, visual rules, templates, reuse.

1 Introduction

Model transformations are key to the success of model-based software development [1]. Model transformations have been applied in significantly diverse use cases as in (a) transforming XML documents from an XSLT representation into an XQuery representation [2], (b) middleware quality of service (QoS) configuration [3], which involves automatically mapping application-specific QoS requirements onto the correct QoS configuration options for the middleware platform, (c) transforming Simulink/Stateflow models into their hybrid automata representation for formal verification [4], and (d) synthesizing dialogs for communication endpoints (*e.g.*, hardware devices/software applications for communications, such as cellphone, instant messenger (IM), pager) in enterprise workflows for rapid decision making [5].

Despite the diversity in application, a noticeable trait is the commonality among model transformations from a similar domain or product line. For example, in the QoS configuration use case, many middleware configurations are the same across a class of applications that are related to each other due to similarities in their QoS requirements and the implementation platform. Similarly, in the dialog use case, despite differences in the communication endpoints, a number of dialog properties remain common.

Our study of the advances in model transformations suggests that despite the strong evidence of recurring patterns in the transformations, model transformation tools and

⋆ Contact Author: a.gokhale@vanderbilt.edu

techniques [6–9] lack support for reusability, modularization and extensibility of the model transformation rules and algorithms. These shortcomings force the transformation developers to reinvent the transformation steps and the translation logic leading to significant code duplication in the transformations, and increased effort in code maintenance and evolution activities.

Recent research efforts [10–12] have demonstrated the use of model transformations to families of applications or product lines [13]. Yet, the following questions remain to be resolved: **(a) *Invariants*:** How can the commonalities in the transformation process be factored out such that they can be reused by the entire application family? **(b) *Variability*:** How can the variabilities be decoupled from the model transformation rules while maximizing the flexibility of the transformation process? **(c) *Extensibility*:** How can the model transformation process for an application family be extended with new variants, with minimally invasive changes to the transformation rules? **(d) *Minimal Intrusiveness*:** How can all of these capabilities be achieved with minimal changes, if any, to existing model transformation tools?

In this paper we present *MTS (Model-transformation Templatization and Specialization)* to address these questions in the context of graphical model transformation tools. MTS provides transformation developers with a simple specification language to define variabilities in their application family such that the variabilities are factored out and are decoupled from the transformation rules. MTS provides a higher order transformation (HOT) [14] [3] algorithm that automates the synthesis of a family-specific *variability metamodel*, which is used by transformation developers to capture the variability across the variants of an application family. Another HOT algorithm defined in MTS generates the specialized instances of the application family variants. MTS requires minimal changes to the underlying model transformation engine.

The rest of the paper is organized as follows: Section 2 discusses a representative case study to concretely motivate the problem and elicit the challenges; Section 3 describes the MTS solution; Section 4 evaluates our approach in terms of efforts saved; Section 5 compares our work with the existing literature; and Section 6 provides concluding remarks.

2 Need for Reusable Model Transformations

This section highlights the need for a reusable model transformation capability by presenting challenges within a small example. Although our representative example uses exogeneous model transformations (*i.e.*, source and target modeling languages are different) [15], the challenges outlined are valid even in endogeneous model transformations (*i.e.*, source and target modeling languages are same).

2.1 Representative Motivational Example: QoS Configuration Mapping

The following example requires an exogenous transformation which translates domain-specified QoS requirements into the underlying middleware platform-specific QoS configuration options. Figure 1 shows the UML representation of both the source and the target metamodels used in the QoS configuration case study. As shown, the source

[3] Since the transformation(s) themselves become the input and/or output, we refer to the transformation process in MTS as higher order transformations (HOTs).

metamodel contains the following Booleans for server components: (1) `fixed_priority_service_execution` that indicates whether the component changes the priority of client service invocations, and (2) `multi_service_levels` to indicate whether the component provides multiple service levels to its clients.

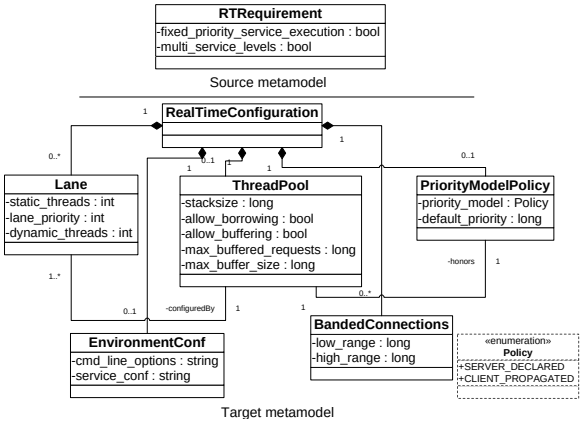

Fig. 1: **A UML Representation of Middleware QoS Configuration Metamodels.**

The target metamodel defines a language to represent real-time CORBA [16] middleware configurations and defines the following elements: (1) `Lane`, which is a logical set of threads, each one of which runs at `lane_priority` priority level. It is possible to configure static threads (*i.e.*, those that remain active until the system is running) and dynamic threads (*i.e.*, those threads that are created and destroyed as required); (2) `ThreadPool`, which controls different settings of `Lane` elements, such as, `stacksize` of threads, whether borrowing of threads across lanes is allowed to minimize priority inversions, and maximum resources assigned to buffer requests that cannot be immediately serviced; (3) `PriorityModelPolicy`, which controls the `ThreadPool` policy model (*i.e.*, whether to serve the request at the client-specified or server-declared priority); and (4) `BandedConnections`, which defines separate connections for individual (client) service invocations to minimize priority inversions.

Transformations for middleware QoS configuration are applicable across a number of application domains. The individual configurations generated using the model transformation should be easily customizable for slight variations in QoS requirements for these domains. Thus, the case study has the following requirements for the generated middleware QoS configurations: (1) the `PriorityModelPolicy` object along with its attributes are transformed from the `fixed_priority_service_execution` source attribute; (2) `Threadpool` and `Lane` objects, and their attributes are transformed from the `multi_service_levels` source attribute. Multiple levels of service indicate multiple priorities that must be handled, which means that a `Threadpool` has multiple `Lane` objects, and that the cardinality and the exact values of their attributes will vary based on QoS requirements. For example, borrowing makes sense for `ThreadPool` only if multiple lanes exist within a thread pool; and (3) whether to configure `BandedConnections` or not may be determined by the developer based on `multi_service_levels` source attribute.

2.2 Impediments to Reusability in Contemporary Model Transformations

This paper focuses on the model transformations that are carried out using model transformation tools, such as ATL [14] and GReAT [9]. These tools conform to a transformation process where the model-to-model transformations are described using transformation rules in either a textual or visual language. These rules relate elements of a source modeling language defined by one metamodel with elements of a destination modeling language corresponding to a different metamodel. The actual transformations are carried out on model instances of the source modeling language, which transform it into model instances belonging to the destination language.

To highlight the impediments to reusability in model transformations and to demonstrate our solution approach, we have chosen the Graph Rewriting And Transformation (GReAT) [9] framework shown in Figure 2. GReAT was developed using the Generic Modeling Environment (GME) [17], which provides a general-purpose editing engine and a separate model-view-controller GUI. GME is metaprogrammable in that the same environment used to define modeling languages is also used to build models, which are instances of the metamodels.

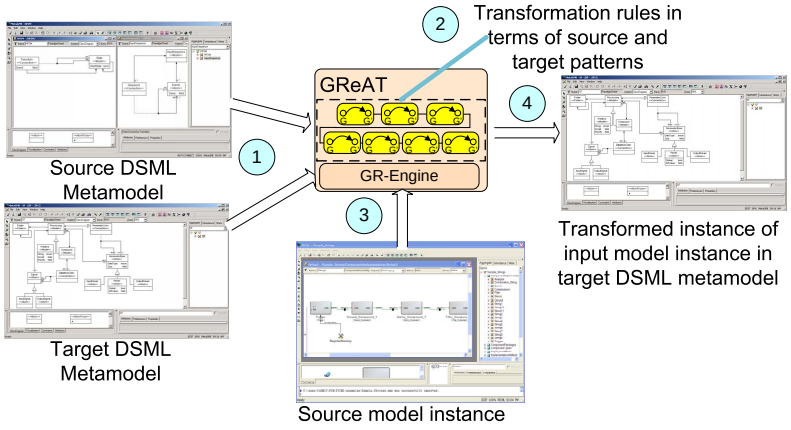

Fig. 2: **Model Transformation Process in GReAT (details of the models are not important for this figure).**

Transformation rules are defined using the GReAT visual language. Figure 2 shows the high-level steps involved in developing transformation algorithms using the GReAT tool chain. In Step 1, the source and target domain-specific modeling languages (DSMLs) for the transformation tool chain are defined. In Step 2, transformation developers use the GReAT transformation language to define various translation rules in terms of patterns[4] of source and target modeling objects. In Step 3, a source model instance is provided to the GReAT framework. Finally, in Step 4, developers execute the GReAT engine (called the GR-engine) that translates the source model using rules specified in Step 2 into the target model.

[4] Here, pattern refers to a valid structural composition using model objects in the source (target) DSML.

Figure 3 illustrates a sample transformation project in our QoS configuration case study. Notice how the entire transformation is composed of a sequence of transformation rule blocks, which can be nested. At the lowest level, a rule block comprises a pattern that describes how one or more elements from the source metamodel must be mapped to one or more elements of the target metamodel. Ports are used to pass objects from one rule block to another block. Rules that cannnot be captured in visual form can be embedded as C++ code in an `AttributeMapping` block.

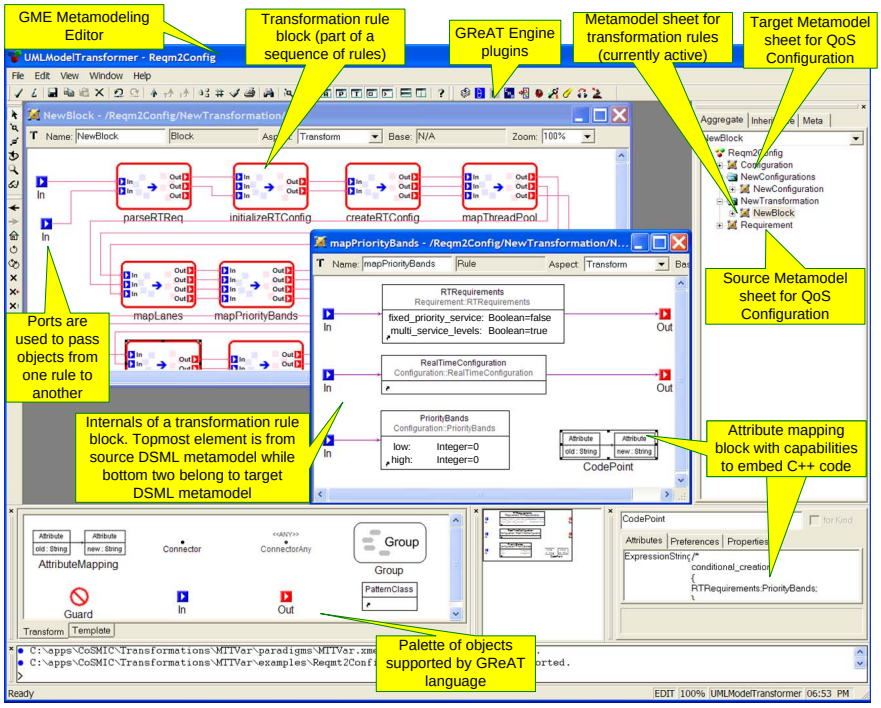

Fig. 3: **Sample Model Transformation for QoS Configurations using GReAT.**

In our case study, different transformations are possible depending on the QoS requirements. For example, the element `BandedConnections` may be present or absent, and the number of `Lane` objects and the priority levels they handle can vary, among other artifacts. However, there exist other transformations that remain invariant. For example, a `Threadpool` object must always be available and by default implicitly always contains one `Lane`.

Due to limited to no support for reuse in contemporary transformation tools, developers are forced to define a complex set of transformation rules (*e.g.*, sequence of rule blocks in Figure 3) for each variant and follow all the transformation steps imposed by the tool. These limitations make a compelling case for reusable model transformations.

3 Templatized Model Transformations

This section presents MTS (Model-transformation Templatization and Specialization). The core idea of MTS is shown in Figure 4. MTS realizes reusable model transfor-

mations in graphical model transformation frameworks using the following four-step approach:

1. Decoupling the variabilities from commonalities: In Step 1 of Figure 4, developers capture the variabilities in transformations in terms of a simple *constraint notation specification* (see Section 3.1). This step decouples the transformation algorithm from its variabilities that can change in an instance-specific manner.

2. Generating variability metamodel: In this step, developers use a higher order transformation (HOT) (*i.e.*, those model transformations that work on meta metamodels to translate source metamodel(s) to target metamodel(s)) defined in MTS to automatically generate the variability metamodel (VMM) for their application family (see Section 3.2). A VMM modularizes the variability in the form of a metamodel, which is the level of abstraction required by the underlying model transformation tool.

3. Synthesizing specialization repository: Next, developers create VMM models, where each VMM model corresponds to an instantiation of the variability for individial family members captured in Step 1. A collection of all the VMM models is termed as a specialization repository for that family (see Section 3.3).

4. Specializing the application instances: Finally, as shown in Step 4, developers use another HOT defined in MTS to create transformation variants (see Section 3.4). This step is similar to instantiating a C++ template where the compiler generates type-specific code based on the type of the argument passed.

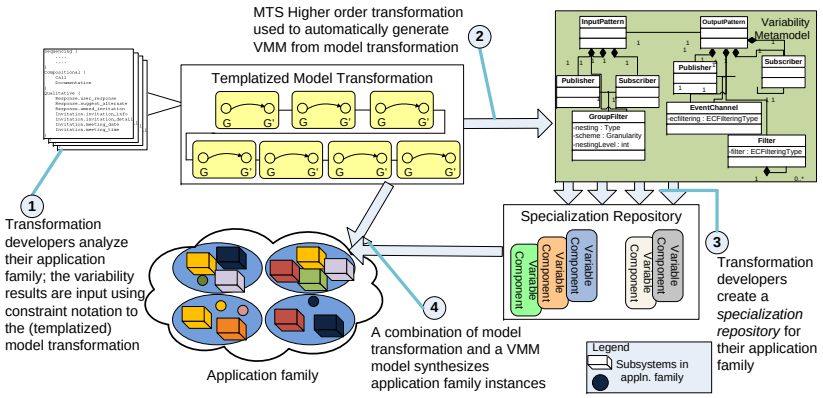

Fig. 4: **MTS Approach to Reusable Model Transformations.**

The remainder of this section provides details of each step, and describes how it supports the four properties (*i.e.*, *Invariants, Variability, Extensibility,* and *Minimal Intrusiveness*) outlined in Section 1.

3.1 Step I: Defining the Templatized Transformation Rules

Section 2.2 elicits the need for reusable model transformation tools to handle variability in model transformations for application families. In programming languages like C++, mechanisms like parametric polymorphism in the form of templates are provided to handle variability. A parametrized class and/or function is then specialized for a concrete type (*i.e.*, a variant). Our solution in MTS is influenced by C++ templates;

specifically, we seek a solution to support templatized transformations and their specialization. The basic idea is that all the commonalities, which constitute the invariants of an application family, are transformed directly from the input models as family *instance-independent* transformation rules using the existing approach of defining the transformation rules. The variabilities are dissociated from the transformation rules to allow independent evolution of the transformation and its variabilities.

A noticeable trait in the transformations for the QoS configuration example illustrates that variability is incurred in either the type and number of structural elements that appear in the target model, and/or the values that are assigned to the attributes of these structural elements. We leverage this observation and define two types of variabilities for our templatized transformation approach:[5]
(a) Structural variabilities, where the basic building blocks, *i.e.*, model elements, or their cardinalities in every family member model are different. Thus, the variation in family member models emanates from dissimilarities in their structural composition.
(b) Quantitative variabilities, where the family member models may share model elements but the data values of their attributes are different.

Both of these variabilities can be denoted as simple implication relations that can be characterized by one of the following types of associations between source ($s \in S$) and target ($t \in T$) objects where, $P1$ and $P2$ denote patterns of source and target objects:

Association	Definition
one-to-one	injective function: $s \in S, t \in T \; \exists \; f(s) \overset{\alpha}{\longmapsto} f(t) \; : if \; f(s) = f(t) \; then \; s = t$.
one-to-many	$s \overset{\phi}{\longmapsto} t$, where $s \in S$, and $t = \{P2(t_1, t_2, ..., t_n) \mid t_{i=1..n} \in T\}$.
many-to-one	$s \overset{\varphi}{\longmapsto} t$, where $t \in T$, and $s = \{P1(s_1, s_2, .., s_m) \mid s_{j=1..m} \in S\}$.
many-to-many	$s \overset{\varphi}{\longmapsto} t$, where $s = \{P1(s_1, s_2, .., s_m) \mid s_{j=1..m} \in S\}$, and $t = \{P2(t_1, t_2, ..., t_n) \mid t_{i=1..n} \in T\}$.

How are these implication relations to be used by the developers in the transformation rules, and how are the rules to be supported in a minimally intrusive manner by the underlying transformation framework? We address this question by exploiting the C++ code embedding feature provided by GReAT (see CodePoint in Figure 3). In particular, we use C++ comments as a means to capture variabilty. Because these are comments, it has no impact on the execution of the GR-engine.

MTS defines a special syntax called the constraint specification notation that is embedded as C++ comments to capture the relations outlined above for the variabilities. Figure 5 shows an excerpt of the templatized transformation rules for the QoS configuration case study. Notice how the *Structural* block notation captures the structural variability in a transformation rule from the source element, *i.e.*, `RTRequirement:multi_service_level`, to the target element, *i.e.*, `BandedConnection`. The templatized rule will not explicitly model a rule for creating a `BandedConnection`. Instead, only the common parts, which includes the `Lane` and `Threadpool` (not shown) are included in the templatized rule.

However, notice that since the attribute values for `Lane` and `Threadpool` may vary, they are captured separately in the form of a *Quantitative* block notation, which

[5] This observation was true for other use cases we studied. Other forms of variability, such as those based on behavior, are also possible but are not addressed in this paper. They are part of our future investigations.

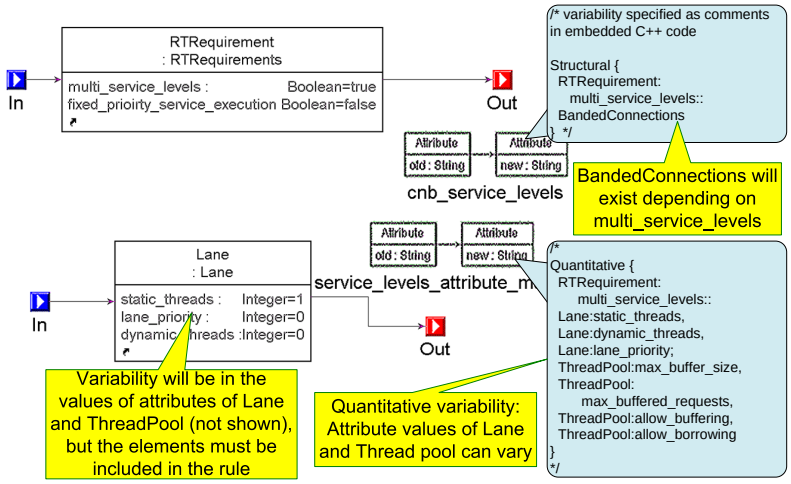

Fig. 5: **Templatized Transformation Rule in QoS Configuration Case Study.**

indicates what all attributes and their values can vary. Since this is a templatization step, the concrete values for these attributes are not mentioned in this step. Section 3.3 explains how this is accomplished.

3.2 Step II: Generating Variability Metamodels from Constraint Specifications

Although the constraint specifications discussed in Step I capture the variability in the transformations, the notation used in the form of C++ comments is oblivious to the model transformation tool. Therefore, it is necessary to make these specifications available at the same level of abstraction as the transformation rules. Because transformation rules associate elements of the source metamodel with that of the target metamodel, MTS defines a HOT to convert the constraint specification into what we call a *Variability Meta Model* (VMM). A VMM essentially modularizes the variabilities and decouples them from the model transformation rules to promote independent evolution.

Algorithm 1 depicts the HOT for generating the VMM. The basic idea behind the algorithm is as follows. Recall from Section 3.1 that the structural variability is concerned only with capturing the (source and target) model objects (or their cardinalities) used in composition of family variants. For every structural variability block, the algorithm creates the corresponding model objects in the VMM. The quantitative variability, on the other hand, captures the dissimilarities in values of model object attributes. Therefore, for these variabilities the algorithm creates model objects and their attributes as well.

The function $initializeVMM(V)$ on Line 3 creates a new VMM, V, and initializes its internal variables. This is necessary so that in the following rules the syntax and semantics of V can be defined in GME. Line 11 reads the source patterns that correspond to every structural variability in the templatized transformation R. Next, the types of each modeling object for the pattern read in the previous rule are deduced by parsing the modeling language as shown in Line 13. This type information is used to create appropriate modeling objects corresponding to the specified source patterns. Similar logic is carried out for patterns in the target language.

After the source and target objects are created in the VMM, in Line 16 the function *composeVariabilityAssociation(V)* creates a simple connection between these objects to denote their association. In a similar fashion, VMM modeling objects are generated for quantitative variabilities in R. Additionally, for quantitative variabilities, attributes of the corresponding modeling objects are also created. The final rule creates a new model object that contains each of these source and target objects created in earlier rules, as shown on Line 20.

Algorithm 1: **Generating VMM from Constraint Specifications.**

```
   Input: source modeling language S, target modeling language T, templatized transformation (set of its rules) R// i.e.,
     the C++ comments encoding the constraint specification
   Output: variability metamodel V
1  begin
2      transformation rule r; constraint notation block cnb; set of constraint notation blocks CNB; structural variability
       cm;set of structural variabilities CM; quantitative variability qm;set of quantitative variabilities QM; pattern p;
       modeling object ob; attribute at; modeling object type type;attribute type atttype; integer c;
3      initializeVMM(V);
4      foreach r ∈ R do
5          if r.cnb() ≠ ∅ then
6              CNB ← r.cnb(); // populate all constraints specifications for that rule
7          foreach cnb ∈ CNB do
8              if cnb.structuralVariabilities() ≠ ∅ then
9                  CM ← cnb.structuralVariabilities();
10                 foreach cm ∈ CM do
11                     p ← cm.SRC();
12                     foreach ob ∈ p do
13                         parseLanguage(S,ob,type); createSRCObject(V,ob,type);
14                     end
15                     /* Do similar steps for patterns in target                                          */
16                     composeVariabilityAssociation(V); /* creates a connection between source and target
                         objects created earlier                                                          */
17                 end
18             if cnb.quantitativeVariabilities() ≠ ∅ then
19                 /* Similarly, create model objects for quantitative variabilities.                      */
20             createContainingObject(V); /* name of the containing object is a combination of rule name, and
                 constraint block name, each of which must be unique                                      */
21         end
22         CNB ← ∅; /* constraint blocks from previous loop are deleted, s.t. those from the next rule can be read */
23     end
24 end
```

We applied Algorithm 1 to the templatized model transformation of our QoS configuration case study to automatically generate a VMM. Figure 6 shows a screenshot of the generated VMM in GME. The variabilities are modeled as pairs of `Source-Pattern` and `TargetPattern`, and annotated by whether they are `Structural` or `Quantitative` using Boolean attributes. The figure corresponds to the `Quantitative` variability rule of Figure 5 in that the attributes of a `Lane` are dependent on the `multi_service_levels` attribute of `RTRequirement`. The `ThreadPool` attribute values can vary among each configuration and are generated in the VMM. The `BandedConnection` element corresponding to the structural variability will be introduced in the VMM in a similar manner using Algorithm 1.

3.3 Step III: Synthesizing a Specialization Repository

In the next step, transformation developers use the generated VMM to create VMM model instances, where each VMM model corresponds to a family member (or more

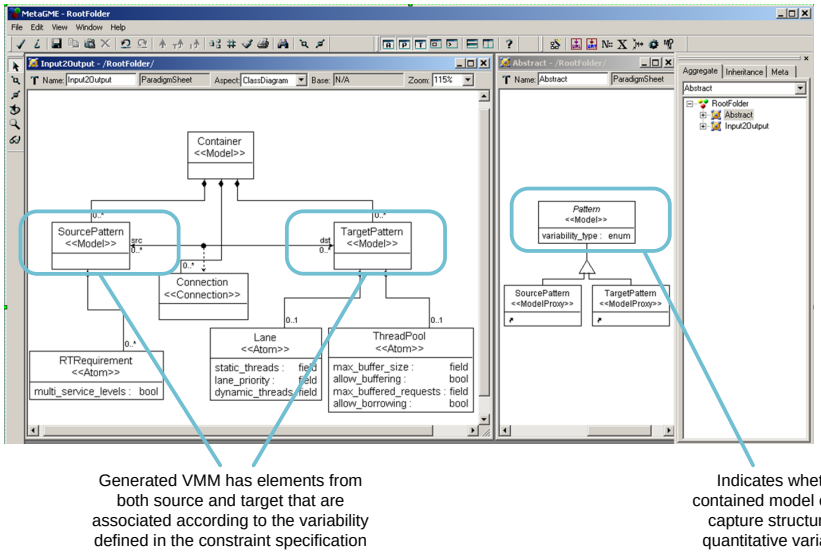

Fig. 6: **Excerpt of Generated VMM for QoS Configuration Study.**

appropriately, variabilities of a family member). A collection of such VMM models for a family is called a specialization repository. This step is akin to the process of providing functors in C++ templates. For example, in a parametrized C++ sort function, programmers are required to supply explicit instantiation of the $\leq$ operator for all the user-defined types for which the sort function is specialized.

Figure 7 shows a sample VMM model that instantiates the quantitative variability in terms of exact values of the `RTRequirement` and `Lane` attributes. Note that because the exact values are now specified as models rather than being encoded in terms of transformation rules, it is considerably easier to modify these values. Additionally, none of the rules are modified to change an existing mapping, and hence the transformation logic need not be re-compiled and linked. This step enables creating as many model instances as the quantitative variability permits without having to modify the rules.

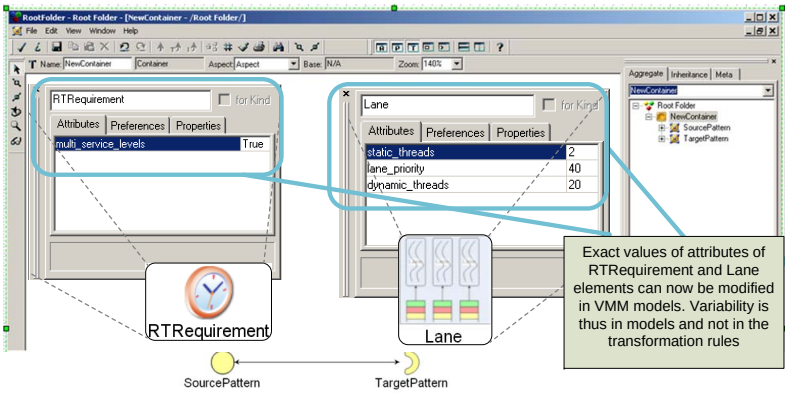

Fig. 7: **A Sample VMM model for a Variant of QoS Configuration Case Study.**

3.4 Step IV: Specializing the Transformation Instances

To realize the actual transformation for a family variant, transformation tools like GReAT must combine the VMM models developed in Step III along with the (original) templatized model transformation rules and the invariant rules. This is akin to a C++ compiler using the programmer-supplied implementation of the $\leq$ operator for a user-defined type and instantiating the remaining parametrized class with the given type. The templatized transformations, however, do not contain any transformation rules for the variability in the visual language. How is it then that the instantiated variability in the form of a VMM model can be recognized by the templatized transformation rules? To address this question, MTS provides a second HOT that (1) reads the input VMM model for a family member, and (2) adds temporary objects at appropriate points in the templatized transformation rules, which serve as placeholders to insert the instantiated variability of a family member (corresponding to the current VMM model).

Algorithm 2 defines the translation rules for a transformation from a VMM model. Lines 3–5 create a new model transformation instance R' from the input templatized transformation R, read the containing model objects in VMM V, and for every model object *ob* search the corresponding rule in the transformation R'.[6] This rule denotes the location where the variabilities contained in *ob* were specified in Section 3.1.

Algorithm 2: **Specializing the Model Transformation from a VMM model.**

```
   Input: variability metamodel V, templatized transformation R
   Output: specialized instance of input templatized transformation R'
1  begin
2      transformation rule r; set of model objects OB,IO; pattern p; modeling object ob,io,tmp; attribute at; modeling
       object type type;attribute type atttype;
3      R' ← R; OB ← containingModelObject(V);
4      foreach ob ∈ OB do
5          r ← searchRule(R',objName(ob)); createTempObject(tmp,r); deleteCNB(r);
6          IO ← parseSRCPattern(ob);
7          foreach io ∈ IO do
8              createObjectRefs(io,tmp); assignCardinalities(io,tmp);
9              createAttribs(io,tmp); assignValues(io,tmp);
10         end
11         /* do similar steps for target pattern                                                          */
12     end
13 end
```

Once rule r is known, the constraint block is deleted from this rule in function $deleteCNB(r)$. The function $createTempObject(tmp, r)$ creates a temporary object tmp inside this rule. For every `Structural` variability in the source pattern in *ob*, object references are read from V, and created in tmp and in addition, their cardinalities are assigned as shown in Line 8. Similarly, attributes in VMM that capture `Quantitative` variabilities are read from V, and created and assigned values in tmp in Line 9. The same rule is also repeated for all target patterns in *ob*.

We applied Algorithm 2 to our QoS configuration example. One of the rules in this case study assigns specific data values to the attributes of `Lane` (target) depending on whether or not the `multi_service_levels` (source) value is set to TRUE. Further,

[6] For creating application family instances, it is not necessary to create a new instance R', but is done only for Algorithm 2 to avoid modification of the original templatized transformation R.

as identified earlier in Section 3.1, there is a quantitative variability involving these two elements. The same variability is also given in Figure 8 for reference. The attributes in `tempObject` are assigned values from the values of the corresponding attribute in the VMM model. Similarly, for the structural variability, the model object references are also created by parsing and reading the VMM model.

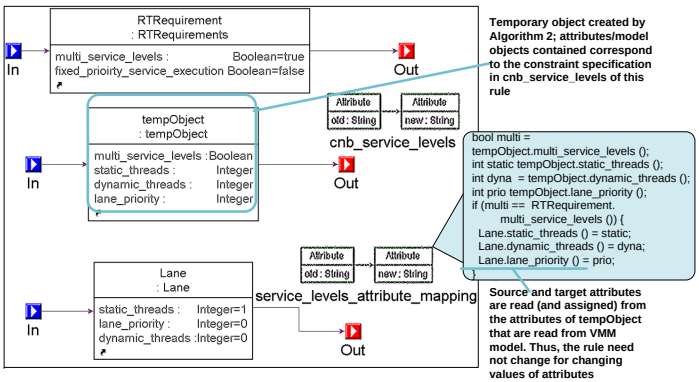

Fig. 8: **Specialization of a QoS configuration rule using MTS.**

Thus, the rule `service_levels_attribute_mapping` (and in effect, the model transformation itself) need not change, when some of these data values/model object cardinalities have to be altered. This is because the modifications can now be done simply by modifying the appropriate VMM model.

4 Evaluating the Efforts Saved using MTS

Because MTS was developed to enhance reusability, this section describes its merits in terms of the reduction in effort to write the transformation rules. The prototype implementation of MTS is part of the CoSMIC[7] tool suite. All of our experiments are based on CoSMIC version 0.5.7, with GME version 6.11.9 and GReAT version 1.6.0.

Recall from Section 3 that to create a target model from a source model using GReAT, developers need to execute the GR-engine that executes all the translation rules of that model transformation. More specifically, developers must first specify all the rules that transform the elements of the source model to the target model. Thereafter, the GR-engine execution involves the following steps: (1) executing the *master interpreter* that generates the necessary intermediate files containing all the rules in the current transformation, (2) compile these intermediate files, if not done already, and (3) run the generated executable. Steps 1 and 2 must be executed each time the model transformation rules are modified – which is the case with variants of a family.

Without MTS, model transformations for each variant of an application family must expend effort in all of the above steps. Our goal is to evaluate MTS in terms of effort saved. We focus on two specific cases discussed below.

Case 1: Newly added variant is subsumed by existing constraint specifications: The existing constraint specification for the application family may be sufficient to capture

[7] CoSMIC is a MDE toolsuite used in the design and deployment of applications for QoS-enabled middleware. It is available from http://www.dre.vanderbilt.edu/cosmic/.

all the variabilities of a new family variant. Thus, the developers can create a new variant simply by re-executing the same model transformation with the VMM model of the variant as one of the inputs to the transformation. Note that the first two steps have to be performed only once when the model transformation is being executed for the first time. Because all the instance-specific customizations/changes are done in the corresponding VMM model, developers only need to execute Step 3 after each change to produce output of the transformation (*i.e.*, a new family instance).

In contrast, the traditional approach of one model transformation per single (subset of) family instance(s) will require maintenance of $I * R_n$ rules, where I is the number of family instances, and R_n is the average number of rules per instance. With MTS, assuming that the average number of rules do not change, the total number of rules to be maintained reduces by a fraction of $\frac{I-1}{I}$.

Case 2: New variant requiring additional constraint specifications: If the variabilities of a new family variant are not completely captured using existing constraint specification for the application family, MTS requires enhancements to the constraint specification itself. Such a change necessitates executing the first two steps above once to produce a new VMM which can be used to model variabilities in the new variant. Note that despite this change, the VMM models corresponding to the existing variants will still be valid provided the changes in constraint specification (because of a new family variant) are orthogonal to the existing variabilities.

5 Related Work

Existing model transformation tools [7, 14, 18] support some form of HOTs. PROGRES and ATL allow specification of type parameters while VIATRA allows development of meta transformations, *i.e.*, HOTs that can manipulate transformation rules and hence model transformations. Unlike MTS, however, these tools do not provide mechanisms for separation of variabilities from model transformations to facilitate automated development of application families.

A recent work synergistic to MTS appears in [12]. In this work the authors propose (1) transformation factorization to extract common parts of two or more transformation definitions into a reusable, base transformation, and (2) composing transformation definitions mapping from a single source metamodel to multiple target metamodels, each representing a specific concern in the system being transformed. MTS differs from [12] in that we focus on composing the common (base) transformation by using the constraint notation (as opposed to factoring out commonalities from existing transformations), and automating the entire process of transformation specialization (*i.e.*, creating instances of transformations).

Reflective model-driven engineering (MDE) [14] proposes a two-dimensional MDE process by expressing model transformations in a tool- or platform-independent way and transforming expressions into actual tool- or platform-specific model transformation expressions. Although reflective MDE focuses on having durable transformation expressions that naturally facilitate technological evolution and development of tool-agnostic transformation projects, mappings still have to be evolved with a change in platform-specific technologies. MTS, however, is concerned with managing and evolving model transformation variability in systems developed using MDE.

Asset variation points discussed in [19] deal with expressing variability in models of product lines [13]. A variation point is identified by several characteristics (*e.g.*, point reference, and context, use and rationale of the variation point) that uniquely identify that point in the product lines. These asset variation points capture variation rules of implementation components of a product line member.

An aspect-oriented approach to managing transformation variability is discussed in [10] that relies on capturing variability in terms of models and code generators. Another approach is model weaving [20], which is used in the composition of separate models that together define the system as a whole. Using the aspect-oriented approach requires developers to learn a new modeling language for creating aspect models for their product line. In contrast, the VMM models generated by MTS use modeling objects that are part of the source (or target) modeling languages requiring no additional learning curve.

6 Conclusions

This paper presented MTS (Model-transformation Templatization and Specialization), which is an enabling technology that seamlessly integrates with existing model transformation tools to support reusable model transformations for application families. MTS defines templatized transformations to factor out the commonalities, and uses the notion of a generated variability metamodel to capture the variabilities in the transformation process across variants of an application family. MTS defines two higher order transformations to specialize the transformations for different variants. Although our existing prototype is implemented in GReAT, it can be extended to other model transformation toolchains if the variabilities can be captured in a non-invasive way as we did in GReAT. The results of evaluating MTS indicate that developer efforts are minimized when new variants are added to the application family, which otherwise require transformation rules to be reinvented in traditional approaches.

Our future work will involve extensive user studies using MTS and quantitatively evaluating efforts saved, as well as measuring the performance overhead of executing the HOTs. MTS is available in open source as part of the CoSMIC MDE tool suite from `www.dre.vanderbilt.edu/cosmic`.

References

1. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software **20**(5) (2003) 42–45
2. Bézivin, J., Dupé, G., Jouault, F., Pitette, G., Rougui, J.E.: First Experiments with the ATL Model Transformation Language: Transforming XSLT into XQuery. In: Companion of the 18^{th} Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003, Anaheim, CA, USA, ACM (2003)
3. Kavimandan, A., Gokhale, A.: Automated Middleware QoS Configuration Techniques using Model Transformations. In: Proceedings of the 14^{th} IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008), St. Louis, MO, USA (April 2008) 93–102
4. Agrawal, A., Simon, G., Karsai, G.: Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. In: International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT), Electronic Notes in Theoretical Computer Science, Barcelona, Spain, Springer-Verlag (March 2004) 43–56

5. Kavimandan, A., Klemm, R., Gokhale, A.: Automated Context-sensitive Dialog Synthesis for Enterprise Workflows using Templatized Model Transformations. In: Proceedings of the 12th International Conference on Enterprise Computing (EDOC '08), Munchen, Germany, IEEE (September 2008) 159–168
6. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Transformation Language. In: Proceedings of the 1^{st} International Conference on Theory and Practice of Model Transformations (ICMT 2008). Volume 5063 of Lecture Notes in Computer Science., Zurich, Switzerland, Springer (July 2008) 46–60
7. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA: Visual Automated Transformations for Formal Verification and Validation of UML Models. In: Proceedings of 17^{th} IEEE International Conference on Automated Software Engineering, Edinburgh, UK, IEEE (September 2002) 267–270
8. Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: International Workshop on Application of Graph Transformations with Industrial Relevance (AGTIVE 2003), Charlottesville, VA, USA (September 2003) 446–453
9. Karsai, G., Agrawal, A., Shi, F., Sprinkle, J.: On the Use of Graph Transformations in the Formal Specification of Computer-Based Systems. In: Proceedings of IEEE TC-ECBS and IFIP10.1 Joint Workshop on Formal Specifications of Computer-Based Systems, Huntsville, AL, USA, IEEE (April 2003) 19–27
10. Voelter, M., Groher, I.: Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In: Proceedings of the 11th Annual Software Product Line Conference (SPLC), Kyoto, Japan (September 2007) 233–242
11. Thomas, F., Delatour, J., Terrier, F., Gérard, S.: Towards a Framework for Explicit Platform-Based Transformations. In: Proceedings of the 11^{th} IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2008), Orlando, FL, USA (May 2008) 211–218
12. Cuadrado, J.S., Molina, J.G.: Approaches for Model Transformation Reuse: Factorization and Composition. In: Proceedings of the 1^{st} International Conference on Theory and Practice of Model Transformations (ICMT 2008). Volume 5063 of Lecture Notes in Computer Science., Zurich, Switzerland, Springer (July 2008) 168–182
13. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, USA (2002)
14. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A Model Transformation Tool. Science of Computer Programming, Special Second Issue on Experimental Software and Toolkits (EST) **72**(1-2) (2008) 31–39
15. Mens, T., Gorp, P.V., Varro, D., Karsai, G.: Applying a Model Transformation Taxonomy to Graph Transformation Technology. In: Proceedings of the International Workshop on Graph and Model Transformation (GraMoT'05). Volume 152 of Lecture Notes in Computer Science., Tallinn, Estonia, Springer-Verlag (September 2006) 143–159
16. Object Management Group: Real-time CORBA Specification. 1.2 edn. (January 2005)
17. Lédeczi, Á., Bakay, Á., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments. Computer **34**(11) (2001) 44–51
18. Schürr, A., Winter, A.J., Zündorf, A.: PROGRES: Language and Environment. In Ehrig, H., Engels, G., Kreowski, H., Rozenberg, G., eds.: Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools, World Scientific Publishing Company (1999) 487–550
19. Salicki, S., Farcet, N.: Expression and Usage of the Variability in the Software Product Lines. In: The 4^{th} International Workshop on Software Product-Family Engineering. Volume 2290 of Lecture Notes in Computer Science., Bilbao, Spain, Springer (October 2001) 304–318
20. Gray, J., Bapty, T., Neema, S.: Handling Crosscutting Constraints in Domain-Specific Modeling. Communications of the ACM (October 2001) 87–93